GETTING STARTED WITH THE ATARI 600 XL

PETER GOODE

Getting Started
with the
ATARI 600XL

Getting Started
with the

ATARI 600XL

Peter Goode

Phoenix Publishing Associates
Bushey, Herts

First published in Great Britain in 1984 by
PHOENIX PUBLISHING ASSOCIATES
14 Vernon Road, Bushey, Herts. WD2 2JL

ISBN 0 9465 7614 9

Printed in Great Britain by
Billing & Sons Ltd., Worcester
Cover design by
Denis Gibney Graphics
Chorleywood, Herts
Typesetting by
Prestige Press (UK) Ltd.,
Chesham, Bucks.

CONTENTS

Introduction

The **ATARI 600XL**, with it's advanced programming, graphics and sound facilities is bound to be a popular computer. All the more because of the support which it enjoys from **ATARI** and independent software and hardware suppliers. Most of this support is, however, aimed at people who have had some previous experience of computing.

This book will help you gain experience with computers, the **ATARI 600XL** in particular, and you will soon begin to understand the meaning of the many buzzwords and strange symbols which those well versed in computers are familiar with.

We start right at the beginning with hints on how to approach your computer at your first meeting and proceed through various chapters to advanced forms of programming. An extensive glossary is included to allow easy reference as you work with your computer.

We hope that this book will make '**getting started with the Atari 600XL**' a pleasant experience. Our thanks are due to the staff of Atari (UK) Ltd, and their public relations company for providing us with the support which is so essential when writing a book of this nature.

1

Starting to use your computer

One of the first things to learn about computers is that they will do exactly what they are instructed to do. Therefore, unless there has been some production fault with a particular computer, when a computer isn't doing what you want it to do it is almost certainly your fault. A large amount of time is spent by all computer programmers (or instructors) in trying to find their mistakes. This process of finding, understanding and removing mistakes (or bugs) in the instructions given to the computer is known as de-bugging.

No home computers are yet available which users can instruct (or program) in any human language. Instead nearly all computers use a language called **BASIC**. This language is incredibly simple to learn, in comparison to a human language such as English. This is because it only uses a few dozen words. **BASIC** is specifically designed to control a computer and is one of the most accessible ways for beginners to computing to control their computer.

From looking at the keyboard of your Atari Computer you will see most of the keys which you would find on a typewriter. There are also some other keys which are not found on typewriters because they would not be required. These keys, such as the metallic keys to the right of the main part of the keyboard, are used to help you to instruct and control the computer.

Before you can use the computer it is necessary to connect it to a television, or monitor. Refer to the booklet which comes with the computer in order to find out how to connect up the television and power supply.

Once the television and power supply are connected to the computer turn on the computer and television. (The computer's power switch is on the back next to the power socket.)

Try pressing keys on the keyboard, you can't damage the computer no matter what you type so just press keys to see what they do. Try typing your name. Notice how the letters that you type appear where the white block was, and the white block moves forward one space. When you get to the edge of the screen the white block moves back to the start of the next line down. This white block is called the cursor. It is important because it shows you where the next letter you type will go.

You have probably noticed that some of the keys which the computer's keyboard has are 'special' keys. These are concerned with controlling the computer.

Try one of them now. Press the key at the bottom right of the main part of the keyboard. It looks like a square which has been divided in two along a diagonal. Once you have pressed this key any normal letters you type will be displayed in 'inverse video'. That is as dark letters on a light background instead of the other way round. In order to get back to normal press the 'diagonal box' key again.

One of the most important keys on the computer's keyboard is the <**RETURN**> key. This is located on the right hand side of the keyboard. When you press it the computer moves the cursor onto a new line. This key also signals to the computer that you have finished typing a command and will

cause the computer to try to obey the command you typed. At this stage though you will probably simply get **ERROR** messages each time you press the <**RETURN**> key. Don't worry about this, it is just the computer's way of telling you that it didn't understand you.

Another useful key is the <**DELETE**> key. Up until now each time you have pressed a key the cursor has been moving forwards and down the screen. When you press the <**DELETE**> key (in the top right of the keyboard) the cursor will move back one space, removing the last letter you typed. This is useful when correcting errors.

For example, suppose you typed:

HELLO THEIR

and realised that you had used the wrong word. What you want to do is to remove the last two letters of **THEIR** and replace them with **RE**.

So, before you press <**RETURN**> you press the <**DELETE**> key twice. This will erase the letters **I** and **R**. Once this is done you can simply press the letters **R** and **E** to finish your correction.

This process is known as editing. It is the same task for which you use liquid paper when writing or typing. However it leaves no mess at all. This is one reason why people use computers in the office, because they can be used to correct mistakes in letters before they are printed, which makes for neater letters.

There are other special keys on the keyboard which you may find useful. It is quite easy to use the computer without knowing how to use them so you can skip to the next heading for now, if you want to try your hand at elementary programming.

The <**RESET**> button, one of the metallic keys to the right of the keyboard, stops the computer doing whatever it was doing and restarts it. This button can be useful if you ever get confused. Simply press it and start again.

The buttons below the **RESET** button are used for controlling cartridges and instructions for their use will be found when you buy Atari cartridge software.

As well as deleting one letter the <**DELETE**> key can also be used to erase everything that you have typed since you last pressed <**RETURN**>. If you type a few letters, then hold down the key marked <**SHIFT**> and press the <**DELETE**> key, everything you have just typed will disappear.

The Atari computer can also produce lower case letters, until now everything you have typed has been displayed in capitals. If you press the <**CAPS**> key, you'll find it on the right hand side of the keyboard, then anything else you type will be displayed in lower case letters. By holding down the <**SHIFT**> key you can get **CAPITALS** once more. In order to get capitals all the time, which you will need for the next section on programming, hold down the <**SHIFT**> key and press <**CAPS**>.

An Experiment with the Computer

Let's see if we can get the computer to do something for us. Turn the computer on and type:

WHAT IS 2 TIMES 3?

As you type, the letters and numbers you have entered will appear on the television screen. However nothing else happens; the computer just sits there. This is why you need one of the most useful words in the language of **BASIC**. It is a special word having no equivalent meaning in a human language.

This word is <**RETURN**>.

Whenever you use this word to the computer it will look at everything which you have typed since the last time you typed <**RETURN**> and it will obey any instructions which it can understand. In order to say <**RETURN**> to the computer you must press the oblong shaped key marked **RETURN** on the right hand side of the main keyboard.

When you press the **RETURN** key the computer will print:

ERROR— WHAT IS 2 TIMES 3?

This is the computer's way of telling you that it doesn't understand the command which you gave it. In other words you have made your first mistake. You have tried to instruct the computer in English and it didn't understand.

Now for your first steps in learning the language **BASIC** which the computer will understand.

The 600XL uses a powerful dialect of **BASIC** known as **ATARI BASIC**.

Try typing:

PRINT 2*3

Don't forget to tell the computer you've finished your instructions by pressing the **RETURN** key.

The computer will print out:

6

READY

underneath your instructions. The number 6 is the result of your command **PRINT 2*3** and the word **READY** means that the computer is ready to do something else. From this you can deduce that the star sign * is somehow understood by the computer to mean times or multiply by.

The computer also understands some other mathematical words in the language of **BASIC**. Some of these are shown below, try typing in these lines, one at a time, remembering to press **RETURN** at the end of each line.

PRINT 1+2+3+4

PRINT 8/28

PRINT (57−23)/(45*(1/4))

You will notice that the computer's answers are about as accurate as a calculators would be. Also notice that the complicated sum doesn't appear to take any longer to work out than the others.

So far we have met two words of the language **BASIC**. These are **PRINT** and **RETURN**. There is another use for the **PRINT** word in **ATARI BASIC** apart from that in printing the results of maths problems. The other use is in printing English words.

Try typing:

PRINT "HELLO COMPUTER"

The computer will obey this command and

HELLO COMPUTER

will be printed.

The computer doesn't understand that you have said hello to it, it is simply obeying your instruction to print **"HELLO COMPUTER"**. In fact any text which you put between the double quote ". . . ." marks after a **PRINT** word will be printed onto the sreen.

The **PRINT** word can also print text and do maths at the same time.

e.g.

PRINT "A NUMBER BIGGER THAN 9 IS ";9+1

will print out:

A NUMBER BIGGER THAN 9 IS 10

but

PRINT "A NUMBER BIGGER THAN 8+1 IS ";9+1

will print:

A NUMBER BIGGER THAN 8+1 IS 10

this is because the text between the quote marks is printed out exactly as it was typed in whereas the computer tries to understand the rest of the line as a statement in the **BASIC** language.

We have now met two more terms of **BASIC**.

Quotes are used in pairs to surround portions of text which we don't want the computer to treat as **BASIC** commands.

The semi-colon is used to tell the computer to print the number immediately after the text preceding the semi-colon.

Another word of **BASIC** is the comma. A comma tells the computer to print the items on either side of it in different columns on the screen. These columns are ten characters wide.

Therefore

PRINT 1;2

will print out

12

whereas

PRINT 1,2

will print

```
1         2
```

This facility can be used to produce neat tables of figures. We will see an example of this later on.

Until now you have not programmed your computer. Instead you have been instructing it in 'real time', i.e., the computer obeys your instructions each time you press <**RETURN**>.

It is essential when using computers for complex tasks, that they don't need to be told what to do in 'real time'. Any speed advantage due to their calculating power would be lost.

The main difference between computers and calculators is that computers can store instructions for use at a later date, whereas calculators (except programmable ones) operate in real time, all the time.

Let's see an example of this, try typing in the following:

```
10 PRINT 12*3
20 PRINT 12*4
30 PRINT 12*5
```

remembering to use a zero, which is at the right hand end of the top row of the keyboard, to make up the numbers at the beginning of each line. Rather than the letter O.

Nothing seems to happen when you press the <**RETURN**> key. In fact something very important does happen. The computer takes the line you have typed in and stores it in a special part of it's memory called 'program memory'.

The computer uses numbers, in the example above they are 10, 20 and 30, to identify and sequence commands which are to be stored in program memory.

If you typed the command:

```
LIST
```

then the computer will print out your 'program'.

```
10 PRINT 12*3
20 PRINT 12*4
30 PRINT 12*5
```

In order to see how the numbers at the start of the lines are used to sequence the lines, type in:

```
25 PRINT " AND FINALLY . . . . ."
```

(remembering to press <**RETURN**>)

Now list the program memory again,

```
10 PRINT 12*3
20 PRINT 12*4
25 PRINT " AND FINALLY . . . . ."
30 PRINT 12*5
```

Notice how the line starting with 25 is inserted between lines 20 and 30. This shows you how Atari Basic orders it's program lines. The numbers at the start of each line are used to indicate to the computer where you want each line to go.

The **LIST** command has some more advanced facilities, apart from simply 'listing' the entire program area on the screen. By adding numbers after the **LIST** command selected parts of the program memory can be listed out.

try typing:

```
LIST 20,30
```

you will see that

```
20 PRINT 12*4
25 PRINT " AND FINALLY ....."
30 PRINT 12*5
```

will be printed out. This is because you have asked for your computer to list everything between and including, 20 and 30

Having typed in what you have been told is a 'computer program' let's try to make it do something.

There is a command in Atari Basic, and indeed in most versions of Basic, which causes the computer to obey the instructions in it's program memory as though they had just been typed in. The instructions are obeyed in numerical order, according to the numbers at the start of each line.

Try typing **RUN** and **<RETURN>**.

The computer should print out

```
36
48
 AND FINALLY
60
```

If you look at the 'listing' of your program you will soon see the relationship between what has been printed out by your program when it is run, and the lines of the program.

Now for one of those neat tables I was talking about.

type in:

NEW

This command tells the computer that we want to type in a new program, so it clears the old one out of it's program memory. If you 'list' program memory now you will see that there is nothing there.

Type in the following program:

```
10 PRINT 1,2,3
20 PRINT 12,
30 PRINT 24,
40 PRINT 36
50 PRINT
60 PRINT "THE TWELVE TIMES TABLE"
```

and '**LIST**' to check that you got it right.

Now type **RUN**, and see if that was what you expected.

See if you can write a program to print the 5 times table from 3 times 5 up to 5 times 5.

2

Programming methods and techniques

A computer program is a set of commands and instructions to the computer which can be saved to tape cassette and used at a later date. Computer programs in **BASIC** are identified by what are known as **'LINE NUMBERS'**. These numbers, when inserted before commands or instructions cause the computer to remember the information following the number and store it in **'Program Memory'**. This is a special area of memory which is used for storing programs.

For example if you type:

PRINT "THIS IS NOT A COMPUTER PROGRAM"

then the computer will print out:

THIS IS NOT A COMPUTER PROGRAM

on the screen. The command which you gave the computer, to **PRINT 'THIS IS NOT A COMPUTER PROGRAM'** was carried out as soon as you pressed return. This is known as **Immediate mode**, because the computer obeyed your command immediately.

In order to make a computer program we first need to tell the computer that we want to start work on a **NEW** program. This is achieved with the **'NEW'** command.

If you type

NEW

then the computer's program memory will be cleared and the computer will be ready to accept a new program. The '**NEW**' command is performed automatically when you turn on your computer.

Now to type in your program, try typing;

10 PRINT "THIS IS A SHORT COMPUTER PROGRAM"

20 END

you have entered a two line computer program. This program contains a new command, **END**. The end command is used to tell the computer where the **END** of your program is. It is not strictly necessary, in this case, to use the **END** command however it is good practise to use it.

The command **LIST**, which you met earlier is used to **LIST** out the contents of the program memory. If you type

LIST

then the computer will print out;

10 PRINT "THIS IS A SHORT COMPUTER PROGRAM"

20 END

the command

LIST 20

would just **LIST** line 20 of the program.

i.e;

20 END

would be printed out.

The computer stores the instructions in it's Program Memory in the order of the line numbers. Suppose that I wanted to add a new line between lines 10 and 20 in the above program.

15 PRINT " A LINE IS INSERTED"

would do the job. The **LIST** command will now cause

10 PRINT "THIS IS A SHORT COMPUTER PROGRAM

15 PRINT " A LINE IS INSERTED"

20 END

to be printed out.

If we thought about it at the time then we could have put both of the **PRINT** commands on line 10 and separated them by a **COLON :** . Atari Basic allows more than one command on each program line, so long as they are separated by Colons.

For example;

10 PRINT"HELLO":PRINT"GOODBYE"

would be an allowed program line and would print out ;

HELLO

GOODBYE

This is the reason why we initially write computer programs with gaps of 10 in the Line Numbers. It leaves plenty of room for inserting new Program lines.

The command which causes the computer to obey the program stored in it's Program Memory is the '**RUN**' command.

RUN

will cause

THIS IS A COMPUTER PROGRAM

A LINE IS INSERTED

to be printed out.

Let's write a simple program

```
NEW

10 PRINT"ROSES ARE RED"
20 PRINT"VIOLETS ARE BLUE"
30 PRINT"THIS LITTLE PROGRAM"
40 PRINT"WAS WRITTEN FOR YOU"
```

Now the command **RUN** will cause this poem program to be **EXECUTED**. When a program is executed the statements are obeyed in the order of the line numbers. That is, in the same order in which the **LIST** command lists them out.

Therefore a **RUN** of the above program will produce

ROSES ARE RED

VIOLETS ARE BLUE

THIS LITTLE PROGRAM

WAS WRITTEN FOR YOU

If you look at the right hand side of the keyboard you will see that the keys – = + and * have also got arrows in boxes printed on them. These keys, when pressed at the same time as the <**CONTROL**> key, which has a box on it to indicate that the boxed parts of the keys will work, move the cursor around the screen. The cursor being, as has been previously mentioned, the block which moves about the screen just in front of the letters and numbers which you type.

By moving the cursor over text which is already on the screen that text can be automatically retyped into the computer. For example type the following;

PRINT "A NON EDITED LINE"

Now move the cursor back over the **PRINT** command by holding down the <**CONTROL**> key and pressing the – (minus) key until the cursor is on top of the 'P' of **PRINT**. Now move the cursor across the **PRINT** command by holding down the <**CONTROL**> key and pressing the * key. (which has a right-arrow on it).

When the cursor is on top of the quote " mark at the end of the line press <**SPACE**> to erase the quote and type

HAS BEEN EDITED"

remembering the quote mark.

If you now press <**RETURN**> then the computer will print out.

A NON EDITED LINE HAS BEEN EDITED

See if you can use the on sreen editing to change our poem program to print

ROSES ARE RED

APPLES ARE GREEN

THIS LITTLE PROGRAM

WAS EDITED ON SCREEN

You could just retype lines 20 and 40, I.e.;

20 PRINT "APPLES ARE GREEN"

40 PRINT "WAS EDITED ON SCREEN"

as the computer would forget it's old lines 20 and 40 of program memory if you give it some new ones. However, it will be good practise to change the original program by Editing.

First of all type **LIST** to get the listing which you are going to change onto the screen. Then use the boxed arrows in conjunction with the <**CONTROL**> key to Edit the program.

FOR / NEXT LOOPS

Suppose that we wanted the computer to print out "**HELLO**" ten times. One way we could do this, with a program would be.

NEW

10 PRINT"HELLO"

20 PRINT"HELLO"

```
30 PRINT"HELLO"

40 PRINT"HELLO"

50 PRINT"HELLO"

60 PRINT"HELLO"

70 PRINT"HELLO"

80 PRINT"HELLO"

90 PRINT"HELLO"

100 PRINT"HELLO"
```

Surely there must be a better way than this, imagine the equivalent program to print **HELLO** ten thousand times. That would not be nice.

Instead the **BASIC** computer language provides us with a much nicer way of doing this. By using a variable, and the commands **FOR** and **NEXT** we can arrange for the computer to print as many **HELLO**'s as we like using only three program lines.

The program to do this is quite small, for example;

```
NEW

10 FOR A=1 TO 10000

20 PRINT"HELLO"

30 NEXT A
```

will print **HELLO** ten thousand times. The **FOR** command instructs the computer to create a variable, in this case the

variable **A**, and to start it off at an initial value of, in this case **1**. The general form of the command is:

FOR A=start TO finish

(some instructions)

NEXT A

the variable **A** will be started off at the value 'start' and will have one added to it each time '**NEXT A**' is encountered, until **A** is equal to, or greater than, '**finish**'. Therefore

```
NEW
10 FOR A=1 TO 5
20 PRINT A
30NEXT A
```

will print out

```
1
2
3
4
5
```

when it is **RUN**.

If we want to, we can make the Computer count in other amounts than ones. For example, if you modify line 10, using screen editing, to read;

10 FOR A=1 TO 9 STEP 2

then

1

3

5

7

9

will be printed out when the program is **RUN**. This is because the number after the '**STEP**' command is the amount which is added onto the variable **A** each time around the loop. We can use this facility to make the computer count backwards.

For example;

NEW

10 FOR COUNT=10 TO 1 STEP -1

20 PRINT "T -";COUNT

30 NEXT COUNT

40 PRINT "BLAST OFF"

will print a countdown when **RUN**.

We can even insert another **FOR NEXT** loop in the middle of the **COUNT** one in order to make a delay. This is because it takes the computer some time to count from **1** to **400** so by adding;

```
24 FOR DELAY=1 to 400

26 NEXT DELAY
```

the countdown can be made to take longer.

INTRODUCING SUBROUTINES

Sometimes when writing computer programs it becomes noticeable that parts of the program are very similar. For example, in the **POEM** program earlier we kept wanting to print out phrases like. "**APPLES ARE GREEN**" or "**ROSES ARE RED**". We could write a **SUB-PROGRAM**, or **SUB-ROUTINE**, whose only purpose is to print out three word phrases with "**ARE**" in the middle. Such a routine could then be 'called in' by the main program to perform it's task whenever it is needed.

The subroutine to do this would look something like this.

```
10000 PRINT A$;" ARE ";

1010 PRINT B$

1020 RETURN
```

A$** and **B$ are string variables holding the two words to be printed either side of the word " **ARE** ". **A$** and **B$** must, of course, be **DIMensioned** with the **DIM** command before they can be used.

The complete poem program might look something like this.

```
NEW

   10 DIM A$(20),B$(20)

   20 A$="BANANAS":B$="YELLOW"

   30 GOSUB 1000

   40 A$="APPLES":B$="GREEN OR RED"

   50 GOSUB 1000

   60 A$="PLUMS":B$="BORING"

   70 GOSUB 1000

   80 PRINT "THIS POEM DIDN'T RHYME VERY
      WELL"

   90 END

 1000 PRINT A$;" ARE ";

 1010 PRINT B$

 1020 RETURN
```

The command **GOSUB** causes the computer to carry on executing the program from the line number following the **GOSUB** command. The **RETURN** command makes the computer **RETURN** to the statement next in the sequence of line numbers after the **GOSUB** command.

It is a good idea to put subroutines at the end of the program on high line numbers, to remind you that they are subroutines of your program and not part of the main program.

It is also a good idea to label your subroutines by using the **REMark** facility of **Atari Basic**. The command **REM** allows any sort of remark to be included on a program line, and ensures that it will not be executed as a part of the program. For instance in the above program we would label the subroutine by putting a **REMark** just before it.

999 REM THE 'THREE WORDS' SUBROUTINE FOLLOWS

STORING DATA WITHIN A PROGRAM

There are a very useful pair of commands in **Atari BASIC** which can be used to store **String** or **Numeric DATA** within programs. The **DATA** is stored in the Program Memory section of the computer's memory. This is achieved with the **DATA** command.

In order to store the three words **'APPLE'**, **'ORANGE'** and **'PEAR'** within your program you would use a program line of the form.

9000 DATA APPLE,ORANGE,PEAR

Note the large line number, it is a standard amongst many programmers to put **DATA** at the end of the Computer Program, after any Subroutines.

You will notice that the items of **DATA** are separated by commas. This is because more than one item of **DATA** has been placed after the **DATA** statement. An equivalent set of program lines would be :

9000 DATA APPLE

9010 DATA ORANGE

9020 DATA PEAR

In order for your program to be able to **READ** the **DATA** it is necessary to use the **READ** command. This command is similar to the **INPUT** command in that it assigns some sort of value to a variable. However instead of stopping program execution and waiting for the program's user to respond, the **READ** command gets it's input **DATA** from **DATA** statements. For instance try typing in :

NEW

9000 DATA 2,4,7

READ A

What number do you think the variable **'A'** has in it now.

PRINT A

should **PRINT** out the number 2

However, if you type

READ A

PRINT A

again, then the number **4** is **PRINTed** out. The **READ** command has **READ** the next item of **DATA**.

If you try the **'READ A'** and **'PRINT A'** sequence again, then you will get the number 7, the last item of **DATA**.

If you try to **READ** any more **DATA** then you will get **ERROR** number **6**. Which is an **'OUT OF DATA'** error. This means that the computer has run out of **DATA**.

In order to be able to **READ** the **DATA** again, we need to use the **RESTORE** command. This will **RESTORE** the pointer

which the computer uses to decide where **DATA** comes from to its original value, and allow us to re-**READ** the **DATA**.

An example program to use the **READ** and **DATA** commands is shown below.

```
10 DIM A$(20),B$(20)

20 FOR COUNT=1 TO 3

30 READ A$,B$

40 GOSUB 1000

50 NEXT COUNT

60 PRINT "THIS PROGRAM'S BETTER"

70 END

1000 PRINT A$;" IS ";

1010 PRINT B$

1020 RETURN

9000 DATA FOG,DAMP

9010 DATA RAIN, WETTER

9020 DATA DO YOU KNOW WHAT IT,
```

and try **RUNning** it.

Notice how the comma at the end of line 9020 fools the Computer into not giving an **'OUT OF DATA ERROR'**.

3

Introducing Variables

Most pocket calculators available today have at least one memory, where numbers can be stored for later use. There is usually a key labelled **'M'** or **'M in'** which allows numbers to be put in this memory.

On the Atari Computer there are a very large number of 'possible' memories. These memories are called **'VARIABLES'**, because they are often used as the variable part of a formula or other arithmetic expression.

For example;

suppose that we wanted to multiply a set of numbers by 863. We could type in the commands in 'real time'.

ie:

```
PRINT 863*24
PRINT 863*612
```

e.t.c..;

or we could write a short program

```
10 PRINT 863*24
20 PRINT 863*612
```

but this doesn't seem to save much time.

Wouldn't it be nice if we could get the computer to ask us for the variable part of the sum, ie: 24,612,e.t.c.. , and then the computer could multiply the numbers we type by 863 and print out the results. As it happens this is very easy in **BASIC**.

First of all we need to understand variables, since we will use one of these to hold the 'variable' part of our calculation.

Type **NEW** to clear the memory of any previous material.

Now type

VAR=10

(not forgetting the <**RETURN**> key after each entry)

followed by:

PRINT VAR

The computer will print out:

10

try

PRINT 4*VAR

and the computer will print

40

as you can see, the computer is substituting the number ten for the word **VAR** each time it finds it in a calculation. This is because the memory called **VAR** was set equal to 10 by the command:

VAR=10

Let's write this into a program

```
10 VAR=10
20 PRINT 863*VAR
```

followed by

RUN

and **<RETURN>**

now all we have to do in order to multiply another number by 863 is to change line 10 of our program.

ie:

```
10 VAR=621
```

after which a **RUN** will give

535923

But this is still not quite what we wanted. We are having to alter the program for each new number. There is a better way which makes use of a new **BASIC** command, **INPUT**.

The command **INPUT** is used to ask Basic to wait, while it is running a program, and accept a number from the keyboard into a variable. In Atari Basic the **INPUT** command also works in real time.

type the following:

VAR=10

PRINT VAR

(**VAR** is 10 as you would expect)

now try;

INPUT VAR

and when a question mark is printed type 73 and press **<RETURN>**

now type in

PRINT VAR

(**VAR** has been changed to 73)

The question mark is produced by **BASIC** when a command is obeyed in order to make it obvious to the person using the computer that they are expected to do something.

Now that we have found how to **INPUT** into a variable in real time, we can try it in a program.

Modify our short program by typing:

```
10 INPUT VAR
```

when the program is **RUN** it should print a question mark and wait for a number to be typed. Try it now.

Your finished program should look like this:

```
10 INPUT  VAR

20 PRINT  863*VAR
```

Now our program can be used for all our multiplying by 863.

Atari Basic is capable of a very large range of arithmetic functions ranging from those useful in games programming to those mainly used for complex mathematics.

Let's try some now:

First of all the maths functions:

PRINT ABS(−24.56)

will print

24.56

this is the **ABSolute** part of **−24.56. ie: its magnitude.**

PRINT LOG(2)

will print

00.69314718

this is the **LOGarithm** of 2, base E, to 8 decimal places.

PRINT CLOG(2)

will print

0.301029995

this is the Common **LOGarithm** of 2, base 10, to 9 decimal places.

PRINT EXP(1)

will print

2.71828179

E to the power 1. This function is only accurate to 6 significant figures although the computer prints it to more.

PRINT SGN(1),SGN(−4.623)

will print

1 −1

this function returns a 1 if the number in the brackets is positive, a Ø if its zero, and a -1 if its negative. Remember that the comma between has set up the spaces.

PRINT INT(5.25461)

will print

5

this function returns the **INTeger** part of the number in brackets. That is to say, the decimals are removed. The computer simply ignores any digits after the decimal point. N.B. No rounding up takes place. That is

INT(4.7) is equal to 4 not 5.

If you want the computer to round up then add Ø.5 before applying the **INT** function.

ie:

INT(4.7+Ø.5) is equal to 5.

PRINT SQR(16)

will print

4

this is the **SQuare Root** of 16. This function always returns the positive root.

There are some more complex mathematical functions such as **ATN** and **SIN** but these will be covered later.

One of the more entertaining functions is **RND**. This function returns a random number between 0 and 1.

PRINT RND(0)

PRINT RND(0)

PRINT RND(0)

will give 3 numbers, which will almost certainly be different, though they **may** be the same since they are random.

The **RND**, or random, function is different from the others in that the number in the brackets doesn't affect the result at all.

A more useful form of random function can be built up from some of the other functions.

PRINT INT(RND(0)*6+0.5)

will give a random number between 1 and 6 inclusive. This is useful for simulating a dice throw.

We can break this down into its component parts.

RND(0) gives a number between 0 and 1.

therefore,

RND(0)*6 gives a number between 0 and 6.

and so;

RND(0)*6+0.5 gives a number between 0.5 and 6.5.

Therefore **INT(RND(0)*6+0.5)** gives a number between 1 and 6. However this number will always be an integer, because of the **INTeger** function.

VARIABLES OF THE SECOND KIND

In the last chapter we met a variable called **VAR**. We saw how to set it equal to any number we liked by use of the = sign.

VAR= − 23.516

is a legal command and would put the number −23.516 into the variable **VAR**.

We could, of course, call our variable almost anything we like. The only major restrictions being that we don't include a **COMMAND** in the variable's name, and that we ensure that the name starts with an alphabetic character.

Therefore:

FRED, MABEL, JOE90

are all legal variable names and can be used as variables in your programs.

However:

PRINT or 9LIVES could not be used.

Try typing in the following program:

```
10 SMALL=0.26
20 BIG 1000
30 PRINT SMALL,BIG
```

```
40 PRINT SMALL*BIG
50 PRINT BIG/SMALL
```

then type **RUN** and <**RETURN**>

```
0.26        1000
260
3846.15384
```

is the expected output.

Because the computer can have a very large number of variables, it can be used to help find the answers to mathematical problems far beyond the reach of most calculators.

As well as these variables which hold numbers, and which are used for mathematical problems, there is another class of variable in the Atari computer.

The string variable.

STRINGS are 'strings' of letters, numbers, punctuation marks or anything else you fancy.

Where numeric variables, which we encountered earlier, hold numbers, string variables can hold almost anything.

String variables, or strings as most computer users refer to them, are used in much the same way as numeric variables. The same rules apply to string variables as regards naming them as apply for numeric variables. However the last character in a string variables name must be a $ (dollar) sign. This is obtained by holding **SHIFT** down and pressing the number 4 on your computer's keyboard.

Before a string variable can be used, the computer must be told its maximum possible length (in characters). This is done with the **DIM** command.

DIM is short for **DIMension**, and the dimension of a **STRING** which the computer needs to know about is its length.

E.g. To tell the computer to expect a **String VARiable** called **SVAR$** with a maximum length of 20 characters use the command:

DIM SVAR$(20)

Where, with a numeric variable you might use the command:

VAR=10.4

the string equivalent type of command would be:

SVAR$="HELLO THERE"

As you will hopefully expect, the command

PRINT SVAR$

would yield:

HELLO THERE

Note that the quotation marks are not stored in the string variable. All that is stored is the information between them.

N.B. If we wish to add two strings together, that is put one on the end of the other, we use the following method.

ABC$="ALPHABET"

EFG$="SOUP"

ABC$(LEN(ABC$)+1)=EFG$

PRINT ABC$

ALPHABET SOUP

this works by asking Basic to make the string variable **ABC$** from its end onwards equal to the string in **EFG$**.

ie:

ABC$ = A L P H A B E T

EFG$ = S O U P

so the **LENgth** of **ABC$** is 8. Being the number of letters in the word **ALPHABET**.

So if we set **ABC$**, to start at **ABC$(9)** equal to **EFG$** then we are effectively adding the word **SOUP** onto the end of the variable **ABC$**.

The string manipulation commands which we used above are very simple to learn. The command is of the form:

ABC$=EFG$(3)

where 3 is a number or a Numeric Variable.

This command will set **ABC$** equal to the character in position 3 in the String Variable **EFG$**. For example if we change **EFG$** as follows

EFG$="ATARI"

then

ABC$=EFG$(3)

would set **ABC$** equal to the letter **'A'** since the third character of **EFG$** is a letter **'A'**.

If a pair of numbers are between the brackets then they are assumed to indicate start and end position within the string. For example

EFG$="COMPUTER"

ABC$=EFG$(4,6)

would set **ABC$** equal to the string **'PUT'**.

N.B. When using these methods to break up strings into smaller parts it is essential that you don't specify parts of the string which aren't there.

For instance

EFG$="SHORT"

ABC$=EFG$(10)

will give an error. (Error 5 in fact.) The error being that you have asked for the 10th letter in **EFG$** which is only 5 letters long!

So it is necessary to take care when using. If you forget to tell the computer about your string variables before you use them, or if you try to make the variable longer than the maximum length you have set, then you will get an error. The computer will print a message of the form:

ERROR– 9

or

ERROR– 9 AT LINE 10

the error number (9) immediately after the word **ERROR** indicates a string error. There is a table of error numbers

together with their meanings and likely causes at the back of this book.

In the same way that numbers can be added, so can strings. But the command for doing this is slightly more complicated. Type the following:

```
DIM ABC$(50),EFG$(100)
```

```
ABC$="HELLO THERE WORLD"
```

so far so good,

we can print the first word,

```
PRINT ABC$(1,5)
```

this asks for characters 1 to 5.

The second word,

```
PRINT ABC$(7,11)
```

this asks for characters 7 to 11 which corresponds to the second word.

Or the last two words,

```
PRINT ABC$(7,17)
```

which asks for characters 7 to 17. **Remember that spaces count as characters.**

There is a special function which tells you the length of a string variable expressed as a number.

Thus the **LENgth** of our string **ABC$** is found by,

PRINT LEN(ABC$)

which should print out

17

Which is, of course, the **LENgth** of **"HELLO THERE WORLD"**.

There are two other important string functions. These are **STR$** and **VAL**.

They are used to convert data stored in numeric variables into string data and vice-versa.

Therefore:

VAR=10.2

EFG$=STR$(VAR)

PRINT EFG$

will print out;

10.2

and similarly:

EFG$="4012"

VAR=VAL(EFG$)

PRINT VAR

will print out;

```
4012
```

These string manipulation facilities are very simple to learn, but their use can often be very complex. Re-read this chapter completely to make sure that you have understood the idea of computers completely.

4

Graphics and Sound

You often see examples of Computer Graphics in everyday life. They are used for special effects by film and television programme makers. Computer graphics are also used for more mundane tasks such as the credits at the end of television programmes. You have also probably met them on 'space invader' type machines. The games cartridges available for the 600XL make extensive use of graphics, as do the programs in **'THE ATARI 600XL PROGRAM BOOK'**.

Computer Graphics include all types of display, from business-like Graphs and Charts through Arcade Games, to Computer Art. The computer can be a powerful tool in the hands of an artist. Also the computer can be made to assist the artist, in much the same way that 'Chord-Organs' help budding musicians. The computer can be programmed to do the tedious work, such as drawing lines and circles, or filling large areas with colour.

The 600XL allows the **BASIC** programmer, i.e. yourself, to use it's very advanced Graphics facilities via a small number of special **'GRAPHICS COMMANDS'**. The first of these commands is the command which tells the computer to switch from text display to graphic. (Up until now the computer has been in **'TEXT'** mode. The following command will put it into one of several Graphics modes.)

GRAPHICS 7

remember to press <**RETURN**> after this, and every command which you give to your computer)

This command tells the 600 XL that we want to use Graphics mode number 7. The 600XL has a total of **13 GRAPHICS** modes, and **3 TEXT** modes. All of which are selected via the **'GRAPHICS'** command.

The different modes available offer varying facilities to the programmer. For instance

GRAPHICS 0

will select the normal **TEXT** mode. This is the mode which is selected when the computer is first turned on. It is also selected whenever the <**RESET**> button is pressed. This is the most convenient mode for typing in programs and commands because it uses only a small amount of memory and can display up to 24 lines of 40 characters each on your television.

However only one colour is available. The two 'apparently' different colours are in fact different shades of the same colour. In order to be able to draw shapes, and to be able to use more than 1 colour, a true **GRAPHICS** mode is required.

Type;

GRAPHICS 7

the background appears to shrink to a small coloured 'window' at the bottom of your television screen. This is known as the **TEXT** window. It is a small portion of 'normal' **TEXT** display at the bottom of an area of the screen which can be used for graphics. Depending upon the graphics mode which has been selected the **GRAPHICS** part of the

screen can be drawn on in up to 16 colours at once. As the number of colours increases the degree of detail which can be shown on the screen decreases.

The buzzword for the degree of detail which can be shown is **RESOLUTION**. Thus a graphics mode which can show a lot of detail is known as a **'HIGH-RESOLUTION'** mode, by the same token a mode which cannot show much detail is known as a **'LOW-RESOLUTION'** mode. The 600 XL has both **HIGH-RESOLUTION** and **LOW-RESOLUTION** modes.

The highest resolution mode on the 600XL is mode 8, in which you draw on a grid of points which has 320 columns and up to 192 rows. However in this mode you can only have one colour in two shades. Similar to the **TEXT** mode.

If you hear a computer 'buff' saying that a particular computer has **'HI-RES'** graphics, then what he means is that the computer in question can show a lot of detail with it's graphics. The 600XL computer has very **HI-RES** graphics.

SIMPLE GRAPHICS

Type

GRAPHICS 7

to put the computer into graphics mode 7.

Graphics mode 7 allows drawing on a grid with 160 columns and 80 rows. Four colours can be used in this graphics mode.

Before starting to draw on the screen we need to tell the computer what colour we wish to draw in. This is done with the **COLOR** command (note the American spelling). For example;

COLOR 1

will allow drawing in orange at the moment. The other colours available are;

COLOR 2 **which is GREEN.**

COLOR 3 **which is DARK BLUE.**

COLOR 4 **which is RED.**

It is also possible to ask to draw in black with the command **COLOR 0. This is often used to erase parts of a drawing on the screen.**

There are two commands for drawing on the 600XL's screen. These are PLOT and **DRAWTO**.

PLOT simply 'plots' one single point at any position on the screen in the currently selected colour.

DRAWTO draws a line from the last point plotted or drawn to, to the point specified. Once again this is done in the currently selected colour.

For example suppose we want to draw a green square.

First of all type

GRAPHICS 7

this will clear the screen.

Then type

COLOR 2

this tells the computer that we want to draw in colour 2, which is at green.

Now, if you type

PLOT 0,0

the computer will plot a point in the top left of the screen. The command

DRAWTO 0,50

will draw a vertical line down the screen. The first number after the **DRAWTO** command is the position of the point to draw to **across** the screen. The second number is the positions **down** the screen. The point 0,0 is in the top left of the screen. The point 159,79 is in the bottom right of the screen.

to complete the square therefore;

DRAWTO 50,50

DRAWTO 50,0

DRAWTO 0,0

COLOUR VARIABLES

In the previous section we drew a square in **GREEN**, which was selected by the command

COLOR 2

This doesn't instruct the computer to draw in green but in **COLOR 2**. It just so happens that the colour **GREEN** is the default colour for **COLOR 2**. We can change the **'ACTUAL'** colour which corresponds to **COLOUR** register 2 with the **SETCOLOR** command.

For example;

SETCOLOR 1,5,6

will change the colour of **COLOR 2** into purple. The setcolor command works as follows;

SETCOLOR variable, colour, shade

where the first number, variable, is the number of a colour variable. Colour variables are similar to ordinary variable in that their contents can be changed. But colour variables must be changed by using the **SETCOLOR** command.

The variable numbers are one less than the corresponding **COLOR**.

I.e;

COLOR 1 **draws in the colour in colour variable Ø.**

COLOR 4 **draws in the colour in colour variable 3.**

The second number of the **SETCOLOR** command chooses the colour which is to be put into the colour variable. These colours are

Colour number :	Actual Colour
Ø	grey
1	light orange (or gold)
2	orange
3	red-orange
4	pink
5	purple
6	purple-blue
7	blue
8	a lighter blue
9	light blue

10	turquoise
11	green-blue
12	green
13	yellow-green
14	orange-green
15	light orange

The final part of the **SETCOLOR** command is the shade, or brightness of the colour selected. Shades vary from 0, which is dark, to 14 which is an intense form of the colour in question. Only even numbers should be used for shades.

To demonstrate the speed with which the **SETCOLOR** command changes colours try the following.

GRAPHICS 7

COLOR 3

PLOT 0,0

DRAWTO 50,60

DRAWTO 100,60

DRAWTO 0,0

this will cause the computer to draw a triangle in dark-blue.

Now to change the colour simply type:

SETCOLOR 2,2,10

The triangle will go orange very quickly.

N.B. **SETCOLOR 2** was used because the triangle was drawn in '**COLOR 3**' and Colour variable 2 corresponds to **COLOR 3**.

AN EXAMPLE GRAPHICS PROGRAM

The following program draws a small house. Try to add to it, to make it draw a front door in a different colour, use **COLOR 1** or **COLOR 4** to choose a different colour.

```
10 GRAPHICS 7

20 COLOR 2

30 PLOT 20,79

40 DRAWTO 20,20

50 DRAWTO 70,20

60 DRAWTO 70,79

70 COLOR 3

80 PLOT 16,22

90 DRAWTO 45,10

100 DRAWTO 74,22
```

ADVANCED GRAPHICS

The 600 XL has many advanced graphics facilities but in order to use them efficiently **BASIC** cannot be used. So it is necessary to program the computer in a language called **'MACHINE CODE'**

This language is not easy to learn and is outside the scope of a 'getting started' book.

These advanced graphics facilities, often called Player/ Missile graphics, are used extensively with machine code, in **THE 600XL PROGRAM BOOK**. From which the following is an example.

PLAYER INSTRUCTION

The missiles will come from the top of the screen and plot their way down towards you. You must cross their path and fire with your SPACE key. Remember that your missiles may take a second to fire so you may have to predict the attacker's path of descent.

To move use keys Z and X for left and right and / and + for down and up.

If you score 5000, or multiples of this number, you will be given a bonus city to defend. As the attackers have different values of points according to which wave you are on you may have to decide to let one of your cities be destroyed in order that you achieve the 5000 points you need to proceed. This may seem a little tough on your citizens but you have to play on, don't you?

Programming Hints

To slow the attackers down, put in a delay statement as follows:

Line 100 FOR QW = 1 to 30: NEXT QW: IF J = 0 THEN 110

The Program

```
10 CI=3:BO=0:SC=0:WV=1
19 DIM X(10),Y(10),A(10),B(10)
```

```
40 GRAPHICS 7+16:POKE 752,1
50 GOSUB 6000
70 N=INT(RND(1)*8+3)
80 GOSUB 3000
90 C=0:M=N
95 IF WV<>1 THEN GOSUB 8000
96 IF CI<3 AND BO<>0 THEN CI=CI+1:BO=BO-
1:GOTO 96
100 IF J=0 THEN 110
106 J=J-1:IF J=0 THEN E=1
110 C=C+1:IF C=80 THEN 700
115 IF N=0 THEN 700
120 FOR L=1 TO M
121 GOSUB 200
122 IF E=0 THEN 125
123 COLOR 2:S=E:X=Q:Y=R:SOUND 0,E*12,4,1
4:GOSUB 4000:E=E+1
124 IF E=21 THEN E=0:SOUND 0,0,0,0
125 IF Y(L)=-1 THEN 190
127 COLOR 3
130 PLOT X(L),Y(L)
140 X(L)=X(L)+A(L):Y(L)=Y(L)+B(L)
145 IF Y(L)>78 THEN Y(L)=-1:N=N-1:GOTO 1
90
150 LOCATE X(L),Y(L),D
160 IF D<>2 THEN 190
170 IF Y(L)>70 THEN GOSUB 7000
180 Y(L)=-1:N=N-1
185 SC=SC+10*WV
190 NEXT L
195 GOTO 499
200 IF PEEK(764)=33 AND E=0 AND J=0 THEN
 J=3:Q=G:R=H
220 IF PEEK(764)=23 AND G>5 THEN COLOR 4
:PLOT G,H:COLOR 2:G=G-1:PLOT G,H
230 IF PEEK(764)=6 AND H>5 THEN COLOR 4:
PLOT G,H:COLOR 2:H=H-1:PLOT G,H
240 IF PEEK(764)=22 AND G<154 THEN COLOR
 4:PLOT G,H:COLOR 2:G=G+1:PLOT G,H
```

```
250 IF PEEK(764)=38 AND H<64 THEN COLOR
4:PLOT G,H:COLOR 2:H=H+1:PLOT G,H
290 RETURN
499 IF CI+BO>0 THEN 100
500 GRAPHICS 17
520 POSITION 0,5
530 PRINT £6;"  score :";SC
540 POSITION 0,12
550 PRINT £6;"     GAME OVER"
560 POSITION 0,7
570 PRINT £6;"  WAVE NUMBER ";WV
580 POSITION 0,19
590 PRINT £6;"   < RETURN >"
600 OPEN £1,4,0,"K:"
610 GET £1,D
620 IF D<>155 THEN 610
630 CLOSE £1:RUN
700 GRAPHICS 17
710 SOUND 0,0,0,0:SOUND 1,0,0,0
720 POSITION 0,5:PRINT £6;"END OF WAVE :
";WV
730 WV=WV+1
740 POSITION 0,10:PRINT £6;"    ";CI;" C
ITIES"
750 POSITION 0,17:PRINT £6;"         bon
us"
760 B=(CI+BO)*100*WV
770 FOR L=B TO 0 STEP -10
775 SOUND 1,L*200/B+40,10,10
780 POSITION 8,19:PRINT £6;" ";L;"   "
790 NEXT L
795 SOUND 1,0,0,0
800 SC=SC+B:IF INT(SC/5000)=SC/5000 THEN
 BO=BO+1:POSITION 6,2:PRINT "BONUS CITY"
890 FOR L=1 TO 800:NEXT L:GOTO 40
1000 COLOR 1
1010 FOR S=1 TO 20
1020 SOUND 0,S*12,4,14:GOSUB 4000
1030 NEXT S
1099 SOUND 0,0,0,0:RETURN
```

```
3000 FOR L=1 TO 10
3020 X(L)=INT(RND(1)*160):Y(L)=0
3025 S=INT(RND(1)*40)+40
3030 ON INT(RND(1)*3+1) GOTO 3040,3050,3
060
3040 A(L)=(25-X(L))/S:B(L)=(78-Y(L))/S:G
OTO 3070
3050 A(L)=(65-X(L))/S:B(L)=(77-Y(L))/S:G
OTO 3070
3060 A(L)=(135-X(L))/S:B(L)=(75-Y(L))/S
3070 NEXT L
3080 G=80:H=50:COLOR 1:PLOT G,H
3090 J=0:E=0:RETURN
4000 SS=S:IF S>10 THEN COLOR 4:SS=S-10
4010 ON SS GOTO 4020,4030,4040,4050,4060
,4070,4080,4090,4100,4110
4020 PLOT X,Y:PLOT X+1,Y:PLOT X,Y+1:RETU
RN
4030 PLOT X-1,Y:PLOT X,Y-1:RETURN
4040 PLOT X+2,Y:PLOT X+1,Y+1:PLOT X,Y+2:
PLOT X-1,Y+1:RETURN
4050 PLOT X-2,Y:PLOT X-1,Y-1:PLOT X,Y-2:
PLOT X+1,Y-1:RETURN
4060 PLOT X-2,Y+2:PLOT X-2,Y+1:PLOT X-3,
Y:PLOT X-2,Y-1:PLOT X-2,Y-2:PLOT X-1,Y-2
4065 PLOT X,Y-3:PLOT X+1,Y-3:PLOT X+1,Y-
2:RETURN
4070 PLOT X+2,Y-2:PLOT X+2,Y-1:PLOT X+3,
Y:PLOT X+2,Y+1:PLOT X+2,Y+2:PLOT X+1,Y+2
4075 PLOT X,Y+3:PLOT X-1,Y+3:PLOT X-1,Y+
2:RETURN
```

```
4080 PLOT X,Y-4:PLOT X-1,Y-4:PLOT X-1,Y-
3:PLOT X-2,Y-3:PLOT X-3,Y-2:PLOT X-3,Y-1
:PLOT X-4,Y-1
4085 PLOT X-4,Y:PLOT X-3,Y+1:PLOT X-3,Y+
2:PLOT X-2,Y+3:PLOT X-1,Y+4:RETURN
4090 PLOT X,Y+4:PLOT X+1,Y+4:PLOT X+1,Y+
3:PLOT X+2,Y+3:PLOT X+3,Y+2:PLOT X+3,Y+1
:PLOT X+4,Y+1
4095 PLOT X+4,Y:PLOT X+3,Y-1:PLOT X+3,Y-
2:PLOT X+2,Y-3:PLOT X+1,Y-4:RETURN
```

```
4100 PLOT X,Y-5:PLOT X-1,Y-5:PLOT X-2,Y-
4:PLOT X-3,Y-3:PLOT X-4,Y-2:PLOT X-5,Y-1
:PLOT X-5,Y
4105 PLOT X-5,Y+1:PLOT X-4,Y+1:PLOT X-4,
Y+2:PLOT X-3,Y+3:PLOT X-2,Y+4:PLOT X-1,Y
+5
4107 RETURN
4110 PLOT X,Y-5:PLOT X+1,Y+5:PLOT X+2,Y+
4:PLOT X+3,Y+3:PLOT X+4,Y+2:PLOT X+5,Y+1
:PLOT X+5,Y
4115 PLOT X+5,Y-1:PLOT X+4,Y-1:PLOT X+4,
Y-2:PLOT X+3,Y-3:PLOT X+2,Y-4:PLOT X+1,Y
-5
4117 RETURN
6000 COLOR 1
6010 RESTORE 10000
6015 READ D:PLOT 0,D
6020 FOR L=10 TO 150 STEP 10
6030 READ D:DRAWTO L,D:NEXT L
6040 READ D:DRAWTO 159,D
6045 COLOR 2
6047 ON CI GOTO 6070,6060,6050
6050 X=20:Y=78:GOSUB 6500
6060 X=60:Y=77:GOSUB 6500
6070 X=130:Y=75:GOSUB 6500
6499 RETURN
6500 RESTORE 11000
6520 PLOT X,Y
6530 READ A,B
6540 DRAWTO X+A,Y+B
6550 X=X+A:Y=Y+B
6560 IF A<>0 OR B<>0 THEN 6530
6599 RETURN
7000 IF X(L)<35 THEN X=25:Y=78:GOSUB 100
0:CI=CI-1:X=20:COLOR 4:GOSUB 6500:GOTO 7
099
7020 IF X(L)<75 THEN X=65:Y=77:GOSUB 100
0:CI=CI-1:X=60:COLOR 4:GOSUB 6500:GOTO 7
099
7030 X=135:Y=75:GOSUB 1000:CI=CI-1:X=130
:COLOR 4:GOSUB 6500:GOTO 7099
```

```
7099 RETURN
8000 C1=INT(RND(1)*16)
8020 C2=INT(RND(1)*16):IF C2=C1 THEN 802
0
8030 C3=INT(RND(1)*16):IF C3=C2 OR C3=C1
 THEN 8030
8040 C4=INT(RND(1)*16):IF C4=C3 OR C4=C2
 OR C4=C1 THEN 8040
8050 C5=INT(RND(1)*16):IF C5=C4 OR C5=C3
 OR C5=C2 OR C5=C1 THEN 8050
8060 SETCOLOR 1,C2,INT(RND(1)*10+5)
8070 SETCOLOR 2,C3,INT(RND(1)*10+5)
8080 SETCOLOR 3,C4,INT(RND(1)*10+5)
8090 SETCOLOR 4,C5,INT(RND(1)*16)
8099 RETURN
10000 DATA 72,75,79,79,73,76,78,78,74,71
,72,74,79,76,76,73,70
11000 DATA 9,0,0,-3,-1,0,0,-2,-3,0,0,3
11010 DATA -2,0,0,-2,-2,0,0,3,-1,0,0,1
11020 DATA 3,0,0,-2,4,0,0,1,1,0,0,-1
11030 DATA 0,0
```

INTRODUCTION TO SOUND

The facility to produce sound and musical effects are often found on home computers, and the 600XL has exceptional abilities in both these fields. Up to four notes can be played at once, over a wide range of notes, through the T.V. loudspeaker.

Sound effects can be useful for alerting users of the computer, or to inform them that their latest Arcade Game has just defeated them again. Musical effects can be used to play tunes and attract people to the computer.

Let's tell the computer to make a sound for us. This is done with the **SOUND** command. It's used like this

Type;

```
SOUND 0,126,10,15
```

This will make the computer produce a middle 'c' note.

Turn up the T.V. volume and (if necessary) tune the computer in until the sound is clear and fairly loud.

If you want to stop the sound type

```
END
```

(and press **<RETURN>**

This will stop any sounds which the computer is currently making.

The sound command is fairly simple to underatand;

```
SOUND a , b , c , d
```

Produces a note on one of the four sound channels which the 600XL has.

The first number after the **SOUND** command (a) is the number of the sound channel which you want the computer to make the sound on. This number must be either Ø, 1, 2 or 3. The same sound will be heard no matter which channel is used. However, in order to play chords it is necessary to use different channels for the diffent notes of the chord.

The second number (b) is the period of the sound. The lower this number the higher the pitch of the sound produced. This number must be between Ø and 255, the numbers Ø and 255 can be used as well. For instance

SOUND Ø,1ØØ,1Ø,15

will produce a higher note than

SOUND Ø,2ØØ,1Ø,15

The third number (c) sets the distortion of the sound produced. This number must be between Ø and 14, it must also be even. Distortions of 1Ø or 14 produce very '**PURE**' tones. Whereas a number such as 4 produces a very harsh tone indeed. The harsh tones tend to be more useful for special sound effects, such as explosions (see the next section), rather than for making nice music.

The last number (d) sets the volume of the note produced by the **SOUND** command. This number must be in the range Ø to 15. The greater this number the louder the note produced. With a volume of Ø the sound is '**OFF**' and will not be heard.

The following program allows you to play notes and chords on your 600XL. Once you have typed it in press the keys on rows 'Q' and 'A' to play the melody. The numeric keys select the chords.

The <**SPACE**> bar will cut off the chord and the <**RETURN**> key will cut off the melody.

The numeric keys represent the chords. Press a number to select.

The keys on rows Q and A represent the 'melody' keys. If you wish to pause during your recital, but don't wish to hear the continuous tone of your last note, press RETURN. Similarly, if you wish to perform a cadenza within your recital and cut off the chord, press SPACE.

We wish you, and your long suffering relatives, many happy hours.

The program

```
10 REM CHORD ORGAN
100 K=PEEK(764)
110 IF K=31 THEN SOUND 1,251,10,10:SOUND
 2,194,10,10:SOUND 3,164,10,10
120 IF K=30 THEN SOUND 1,162,10,10:SOUND
 2,128,10,10:SOUND 3,108,10,10
130 IF K=26 THEN SOUND 1,217,10,10:SOUND
 2,173,10,10:SOUND 3,144,10,10
140 IF K=24 THEN SOUND 1,144,10,10:SOUND
 2,114,10,10:SOUND 3,96,10,10
150 IF K=29 THEN SOUND 1,193,10,10:SOUND
 2,153,10,10:SOUND 3,128,10,10
160 IF K=27 THEN SOUND 1,128,10,10:SOUND
 2,102,10,10:SOUND 3,85,10,10
170 IF K=51 THEN SOUND 1,187,10,10:SOUND
 2,148,10,10:SOUND 3,124,10,10
180 IF K=53 THEN SOUND 1,251,10,10:SOUND
 2,204,10,10:SOUND 3,164,10,10
190 IF K=48 THEN SOUND 1,162,10,10:SOUND
 2,136,10,10:SOUND 3,108,10,10
200 IF K=50 THEN SOUND 1,217,10,10:SOUND
```

```
 2,184,10,10:SOUND 3,144,10,10
210 IF K=33 THEN SOUND 0,0,0,0:SOUND 1,0
,0,0:SOUND 2,0,0,0:SOUND 3,0,0,0
220 IF K=12 THEN SOUND 0,0,0,0
230 IF K=63 THEN SOUND 0,124,10,14
240 IF K=46 THEN SOUND 0,114,10,14
250 IF K=62 THEN SOUND 0,108,10,14
260 IF K=42 THEN SOUND 0,102,10,14
270 IF K=58 THEN SOUND 0,96,10,14
280 IF K=56 THEN SOUND 0,91,10,14
290 IF K=45 THEN SOUND 0,85,10,14
300 IF K=61 THEN SOUND 0,81,10,14
310 IF K=43 THEN SOUND 0,76,10,14
320 IF K=57 THEN SOUND 0,72,10,14
330 IF K=11 THEN SOUND 0,68,10,14
340 IF K=1 THEN SOUND 0,64,10,14
350 IF K=5 THEN SOUND 0,61,10,14
360 IF K=8 THEN SOUND 0,57,10,14
370 IF K=0 THEN SOUND 0,53,10,14
380 IF K=10 THEN SOUND 0,50,10,14
390 IF K=2 THEN SOUND 0,47,10,14
400 IF K=6 THEN SOUND 0,45,10,14
410 IF K=15 THEN SOUND 0,42,10,14
420 IF K=7 THEN SOUND 0,40,10,14
430 GOTO 100
```

SOUND EFFECTS ROUTINES

There follows a selection of sound effects routines which you make like to include in your own programs. Don't worry if you can't immediately understand how the routines work. Study them slowly and all will become obvious.

```
1000REM EXPLOSION ROUTINE
1010FOR L=0 TO 250 STEP 3
1020SOUND 0,L,4,15-L/17
1030NEXT L
1040RETURN
```

```
2000REM PHASER ROUTINE
2010FOR L=3 TO STEP -1
2020FOR I=0 TO 150 STEP 7
2030SOUND 0,I,10,L*5
2040NEXT I
2050NEXT L
2060RETURN
```

```
3000REM SPACESHIP ALARM

3010FOR L=1 TO 8

3020FOR I=100 TO 0 STEP -6

30303SOUND 0,I,10,15

3040SOUND 1,i+5,10,15

3050NEXT I

3060NEXT L

3070SOUND 0,0,0,0

3080SOUND 1,0,0,0

3090RETURN
```

5

Advanced Programming

Up until now the programs which we have written have all performed fairly simple tasks. None of the programs have had to make 'decisions'. It is essential that we discover some method in which we can program the computer to do different things, depending upon what happens whilst the Computer Program is **RUNning**.

In **Atari Basic** there is a very simple method of controlling the responses which the computer makes. This control is achieved with the **IF / THEN** pair of statements. Their use is illustrated :

```
NEW

10 INPUT A

20 IF A=1 THEN PRINT"YOU TYPED THE DIGIT
1"

30 IF A<> THEN PRINT"YOU DIDN'T TYPE A 1"

40 GOTO 10
```

When this short program is **RUN** it will wait for you to type a number and press **<RETURN>**. Then, depending on whether you typed a **'1'** or not the computer will either print out :

YOU TYPED THE DIGIT 1

or, if you didn't type a 1,

YOU DIDN'T TYPE A 1

N.B. the <> pair of symbols are used to mean '**NOT EQUAL TO**' in **ATARI BASIC**.

The general form of an **IF** statement is ;

IF condition THEN statement

Where the condition can be a simple expression such as

A=1

or a complex one such as

A=INT(A/2)*2

IF the condition is true **THEN** the statement will be executed. However, if the condition is not true, i.e.; **IF** it is false, then the statement will not be executed.

For instance the following program will tell you whether you typed an even or odd number.

```
NEW

10 INPUT A

20 IF A=INT (A/2)*2 THEN PRINT"YOU TYPED AN EVEN NUMBER":GOTO 40

30 PRINT"YOU TYPED AN EVEN NUMBER"

40 GOTO 10
```

Notice how the 'statement' part of the **IF** command can be more than one statement, the individual statements being separated by colons : .

The **IF** command can also be used with string variables as part of the condition. For example, the following program will ask you to spell a word and will use the **IF** statement to see if you are right.

```
NEW

 10 DIM A$(20),B$(20)

 20 A$="PROGRAM"

 30 REM NOW CLEAR THE SCREEN

 40 GRAPHICS 0

 50 PRINT"SPELL";A$

 60 REM NOW WAIT FOR A SHORT TIME

 70 FOR DELAY=1 TO 500

 80 NEXT DELAY

 90 REM NOW CLEAR THE SCREEN AGAIN

100 GRAPHICS 0

110 PRINT"WHAT WAS THE WORD";

120 INPUT B$

130 IF A$=B$ THEN PRINT"CORRECT . . . . WELL
DONE":GOTO 150
```

```
140 PRINT"WRONG, THE WORD WAS";A$

150 END
```

It is possible, in **Atari Basic**, to have a whole set of variables called by the same name. These variables must however be Numeric, that is Strings cannot be used. In order to tell the computer that we want a Set, or **ARRAY**, of variables with the same name the **DIMension** command is used. This is the same command which was used to tell the computer the maximum length of a string variable. In this case it is used to tell the computer how many variables we want to be called by the same name. The use of the **DIM** command is similar. I.e.;

DIM MONTHS(12)

will create an **ARRAY** of twelve variables, each called **MONTHS**. The variables in the **ARRAY 'MONTHS'** are accessed separately by the use of sub-scripts. By 'sub-scripting' a number in brackets to the end of the name **MONTHS**, each of the twelve variables called **MONTHS** can be used.

For example,

MONTHS(1)=5

will set the value of **MONTHS** variable number **1** equal to the number 5

MONTHS(4)=6

will set the value of **MONTHS** variable number **4** equal to the number **6**

The values of the variables can be **PRINTed** out in the normal way. E.g.,

PRINT MONTHS(1)

will print the contents of **MONTHS** variable number 1, in this case the number 5.

A **FOR / NEXT** loop could be used to print out the values of all the **MONTHS** variables as follows.

FOR I=1 TO 12:PRINT MONTHS(I):NEXT I

Notice how variables which have not yet been used, except in the **DIMension** command, such as **MONTHS(2)** and **MONTHS(12)**, have been set equal to zero.

So far, when the **PRINT** command has been used it has always **PRINTed** each line of information on the next line down the screen. When the cursor gets to the bottom of the screen the whole screen moves up one line in order to make room for a new blank one at the bottom, where the cursor is. It is often useful when a program is **PRINTing** to the screen, to be able to **POSITION** the cursor wherever you like. This could make our countdown program look better, because the countdown numbers could all be printed in the same place on the screen. Try the following,

Press return a few times to get the cursor away from the top of the screen. Then type

POSITION 0,0

Notice how the cursor jumps back to the top of the screen. The two numbers after the **POSITION** command are the horizontal and vertical distances, in characters, from the top left of the screen which you want the cursor to move to.

The blank line was **PRINTed** by the computer because you had typed a legal command. After most legal commands, which aren't in programs, the computer will **PRINT** a blank line and the word **READY** on the line underneath. It will then let you type on the line underneath the word **READY**.

So an advanced version of our countdown program might look like this :

```
NEW

 10 GRAPHICS 0
 20 POSITION 5,10
 30 PRINT "========="
 40 POSITION 5,11
 50 PRINT ":        :"
 60 POSITION 5,12
 70 PRINT "========="
 80 FOR COUNT=10 TO 1 STEP -1
 90 POSITION 8,11
100 PRINT "   ";COUNT;"   "
110 FOR DELAY=1 TO 500
120 NEXT DELAY
130 NEXT COUNT
140 POSITION 5,11
```

```
150 PRINT "LIFT OFF"

160 FOR ROCKET=1 TO 22

170 PRINT

180 NEXT ROCKET

190 END
```

This is much better than the last version.

There is a useful variation upon the **GOTO** and **GOSUB** commands. When prefixed by the command word **ON** and a numeric expression a selection of parts of the program can be jumped to with **GOTO**, or called as subroutines with **GOSUB**. The **GOTO** or **GOSUB** command is followed by a list of one or more line numbers. Then the line which actually jumped to or called as a subroutine is determined by the value of the numeric expression after the command word **ON**. For example:

```
ON A GOTO 100,200,300,400
```

will execute **GOTO 100**, and continue execution from line 100 of the program, if **A** is equal to **1**. If **A** equals **2** then line 200 will be jumped. **3** selects 300 and **4** would select 400. This form of command can be useful when the programmer wants to give the user of the program a set of several choices. For example:

```
10 GRAPHICS 0

20 PRINT "    MAIN COMMAND MENU"

30 PRINT "  1) NEW USER "
```

```
40 PRINT "  2) ENTER USER'S NAME "

50 PRINT "  3) PRINT USER'S NAME "

60 PRINT "  4) QUIT PROGRAM "

70 PRINT

80 PRINT "  ENTER THE OPTION NUMBER"

90 PRINT "  WHICH YOU WANT "

100 PRINT "  AND PRESS RETURN"

110 INPUT N

120 IF N<1 OR N>4 or N<>INT(N) THEN GOTO
10

130 ON N GOSUB 1000,2000,3000,4000

140 GOTO 10
```

. . . the main part of the program follows . . .

This **MENU** routine will allow several routines to be selected by the program's user via the **ON . . . GOSUB** command.

The expression after the command **ON** can be as complicated as you like so long as it's result is always a positive integer between 1 and the number of choices of line number following the **GOTO** or **GOSUB** part of the command.

We have seen how the **DIMension** command is used to tell the computer how much memory we want to use for Strings and Numeric arrays. Suppose you want to increase the maximum length of a string, or change the size of an array. If

you simply use the **DIM** command again without your program having been restarted with **RUN**, then you will get an error. There is a special command **CLR** which **CLeaRs** any knowledge which the computer has of your arrays and strings and allows you to start again with the **DIM** command.

We saw previously how the **RESTORE** command is used to tell the computer that we want to start **READing** our **DATA** from the beginning again. There is a useful extension to the **RESTORE** command. If you follow it with a line number, e.g.;

RESTORE 450

then the next **READ** command will **READ** the first piece of **DATA** which follows the line number given (in this case line 450). If there is no more data before the end of the program then an error will occur and the program will stop running. Error code **6** will be given which means '**out of data error**'. In other words there is no more **DATA** available to **READ**.

There is a special graphics command, **LOCATE**, which does almost the opposite of the **PLOT** command. Instead of writing information onto the screen it reads information from it into a numeric variable. For example :

LOCATE 40,30,A

will put the Colour number which has been **PLOTted** or **DRAWn** at point 40,30 on the screen into the variable **A**. If used in text modes then the **ATASCII** code number for the character at the relevant position on the screen will be transferred into the variable.

If you are writing a large program on your computer and you want to know how much **FREe** memory the computer has left then you can use the **FRE(0)** function to reveal this valuable piece of information. The command :

```
PRINT FRE(0)
```

will tell you how many bytes of memory are still **FREe** for your use. The number in brackets does not affect the value returned by the function. Each byte of memory can hold one character or digit, or one **BASIC** command word.

There are two functions in Atari basic which are complementary. One of them will return the **ATASCII** code of any character which it is given, the other will return a character for a given **ATASCII** code. These two functions are useful for performing tricks with String variables. For example :

```
100 DIM M$(100),C$(100)

110 PRINT " TYPE IN THE MESSAGE "

120 INPUT M$

125  IF M$="" THEN GOTO 120

130 C$=""

140 FOR L=1 TO LEN(M$)

145 IF M$(L)<"A" OR M$(L)>"Z" THEN
C$(L)=M$(L):GOTO 170

150 C$(L)=CHR$(ASC(M$(L)+1)

160 IF C$(L)=CHR$(ASC("Z")+1) THEN C$(L)="A"

170 NEXT L

180 PRINT "THE CODED VERSION IS"
```

```
190 PRINT C$

200 END
```

This program will encode a message. See if you can write a similar program to decode messages encoded in this manner.
As you are probably aware the **ATARI** range of computers are capable of playing some of the most stunning games available for home computers. Many of these games use joysticks or paddle controllers which plug into sockets to the right of the keyboard. You can use up to four paddles, which can only control movement in one direction, or two joysticks. There are a number of special functions which can be used to determine both the positions of the game controllers, and whether or not the fire buttons are being pressed.

The **PADDLE** function returns a number between 1 and 228 which corresponds to the position of the paddle whose number is given in the brackets. Paddle numbers range from Ø to 3, paddle Ø is the paddle nearest the front of the computer. The function will return 1 if the paddle is fully turned to the right and 228 when fully left. For example:

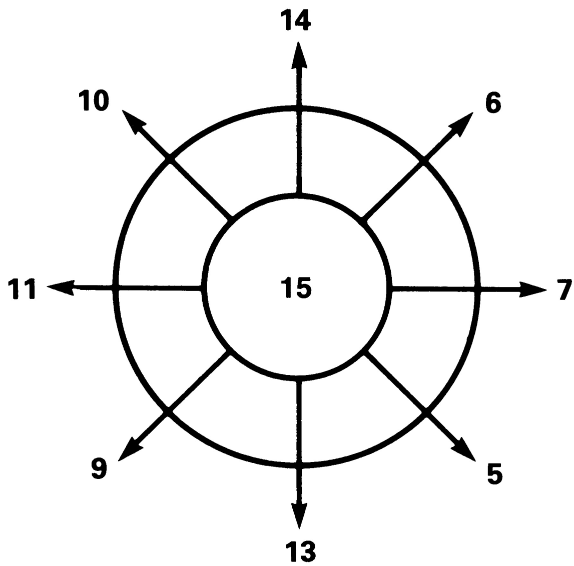
```
PRINT PADDLE(3)
```

will print a number corresponding to the position of paddle number 3.

The **PTRIG** function returns 1 or Ø depending upon whether the trigger of the paddle specified is pressed or not. If the trigger is pressed then a zero will be returned.

Similarly the **STRIG** functions returns a 1 or Ø depending upon whether the joystick fire buttons are pressed or not. Joystick numbers are zero and one. Joystick zero is the front joystick.

The **STICK** function returns a number which corresponds to the position of the joystick indicated. The numbers returned are shown below.



6

Building a Large Program

So far you have met all of the various commands and functions of Atari Basic. If you have in your mind a problem which you think the computer can help you solve, then you now have all the programming tools necessary to start work on a major program.

However, it is a good idea to develop smaller programs first, and to begin to learn the process of debugging from these, before you get involved with really complex programs. Large programs can be very hard and frustrating to debug.

In order to give you some idea of the way in which large programs are developed here is the program **"WIZARDS AND WARLOCKS'** from the **ATARI 600XL** program book, together with a step by step description of what the various parts of the program do.

First of all it is necessary for the programmer to decide in detail what the program should do.

Wizards and Warlocks is a computer based offshoot of the famous role playing fantasy game genre. Unlike many computer adventure games there is no direct task for the player to achieve, other than escaping from the maze in which he/she is trapped.

During the escape the character must fight with many different types of monster. These fights are simulated by the computer using a fairly sophisticated combat system. In essence the amount of damage which players and monsters do to each other is dependant upon their strengths.

Various variables, arrays and string variables will be used by the program to store the information on the monsters and the player. Various subroutines and parts of the main program will, by using and modifying these variables, decide the outcome of fights and uses of magic by the players and monsters.

Now into the program, you may choose to type the program into your computer in the order in which the program lines appear in the text, or if you turn to the end of the chapter you will find a complete listing of the program. Either way you will end up with a working version of Wizards and Warlocks.

The first line in any program should be used to indicate what task the program does. This program plays Wizards and Warlocks and so the first line of the program is . . . .

```
10 REM WIZARDS AND WARLOCKS
```

the **REM** command is used to insert a **REM**ark into the program. The next two lines set up the sizes of the arrays and string variables which the program uses. This is done with the **DIM**ension command.

```
20 DIM N$(80),MG(10),ST(10),FI(10),A$(70
)
25 DIM T$(10),E$(6)
```

then we need to copy all the information about the various strengths of the monsters into these arrays, we also need to copy the names of the monsters into a long string. This task is performed by a subroutine. The subroutine is located at line 1000. It is divided into several distinct sections. The first of these **RESTORES** the data pointer and **READs** in the names of the monsters.

```
1000 REM READ IN CHARACTERISTICS
1001 REM OF MONSTERS
1010 RESTORE :N$=""
1020 FOR L=1 TO 10
1030 READ T$:N$(LEN(N$)+1)=T$
1040 NEXT L
```

the names are built up, one after another, into the string variables **N$**. The next section **READs** in the adjectives which the program uses to describe monsters, into the string variable A$. Notice how each of the monster names is either seven characters long, or is made up to seven characters with spaces. This means that the ten monster names start at character positions 1,8,15,22,29,...etc. in the string variable **N$**. A similar sort of coding is used for the adjectives. It is essential, in order that the names and adjectives are stored in the correct places that the **DATA** lines, from **9000** to **9110**, are typed in correctly with any spaces intact.

```
1050 A$=""
1060 FOR L=1 TO 10
1070 READ T$:A$(LEN(A$)+1)=T$
1080 NEXT L
```

```
9000 DATA ELF     ,DWARF  ,TROLL  ,WIZARD
 ,TOAD   ,WARLOCK
9010 DATA ZOMBIE ,BEETLE ,ORC    ,CHIMER
A
9100 DATA CRUEL ,GREEDY,FOUL  ,WICKED,SL
IMY ,SMELLY,CLEVER
9110 DATA LARGE ,SLY   ,HONEST
```

The final section of the '**READ IN CHARACTERISTICS**' subroutine **READs** in the magic, strength and preferred attack numbers for each of the ten monsters. These numbers are stored in the arrays **MG, ST** and **FI.** The **DATA** which is **READ** in is located on lines **9200** to **9290**, the last line of the program

```
1100 FOR L=1 TO 10
1110 READ A,B,C
1120 MG(L)=A:ST(L)=B:FI(L)=C
1130 NEXT L
```

```
9200 DATA 5,5,10
9210 DATA 1,8,70
9220 DATA 0,15,100
9230 DATA 20,7,5
9240 DATA 0,20,100
9250 DATA 0,3,100
9260 DATA 3,10,65
9270 DATA 0,5,100
9280 DATA 2,8,50
9290 DATA 10,40,50
```

the subroutine ends with the **RETURN** command at line **1199** which sends the computer back to line **40** of the program.

```
1199 RETURN
```

```
30 GOSUB 1000
40 GRAPHICS 0
```

lines 40 to 80 of the program set up the values of variables such as the level of the maze on which the character is stuck, the players score, and strength, magic and health numbers for the player. Line 40 clears the screen and selects a text type of display, mode number 0. Line 50 **OPEN**s a channel to the keyboard for input and will subsequently allow the **GET #1** command to read the keyboard. This printer shows # sign as **£1.**

```
50 OPEN £1,4,0,"K:"
60 LV=6:REM STUCK ON SIXTH LEVEL
70 SC=0
80 ST=10:HL=ST+5:MG=2
```

The next line of the program is another **REMark** and indicates where the 'player has just wandered into a new location of the maze' routine starts.

```
99 REM NEW LOCATION
```

This section of code starts by setting the string variable **E$** equal to a line of six spaces. **E$** is used to hold the single letters corresponding to legal Exits from individual locations. These letters are **N,S,E,W,U** and **D**. Which stand respectively for North, South, East, West, Up and Down. A little later on there is a section of code which, at random, decides which directions the player can move in.

```
100 E$(1,6)="      "
```

```
110 IF RND(1)>0.7 THEN E$(1)="N     "
120 IF RND(1)>0.7 THEN E$(2)="E    "
130 IF RND(1)>0.7 THEN E$(3)="S   "
140 IF RND(1)>0.7 THEN E$(4)="W  "
150 IF RND(1)>0.95 THEN E$(5)="U "
160 IF RND(1)>0.95 THEN E$(6)="D"
165 IF E$="      " THEN 110
```

Line 165 will repeatedly loop back to line 110 until the location has at least one exit.

Line 101 increases the players score, because he has survived one more location. Line 102 checks to see if the player has escaped, which occurs when the variable **LV** is equal to zero. **LV** is a variable storing the number of levels of maze below the surface the player has reached.

```
101 SC=SC+1
102 IF LV=0 THEN 900
```

Then we set the variable **M** equal to **zero**. **M** is used to indicate whether or not there is a monster in the vicinity of the player. **M** is what programmers often call a flag variable. It only has two possible values during the execution of the program. When **M=0** there is no monster, and when **M=1** there is a monster. This **'FLAG'** is used by various parts of the program to find out if a monster is present.

Another thing which happens whilst a player wanders about is that he slowly heals and gains in magical power. Line 107 increases the variable **MG** by one, which is a measure of the **MAGICal** power of the player. Line 109 increases the players health if it has been reduced at all.

```
105 M=0
107 MG=MG+1
109 IF HL<ST+5 THEN HL=HL+1
```

After the section from lines 110 to 165 which sets up the exit string variable **E$** to indicate legal exits from the room the program begins to **PRINT** things. It starts by informing the player that he is in a new room. This is done by line 170.

```
170 PRINT "YOU ARE IN A NEW ROOM"
```

Then the program **PRINTs** a message informing the player of the legal exit directions. This is done by lines 180 to 200. The extra **PRINT** commands leave gaps in the screen display, which makes it easier to read.

```
180 PRINT :PRINT "THERE ARE EXITS :"
190 PRINT "     ";E$
200 PRINT
```

Line 210 decides randomly whether or not there should be a monster in the room. Since **RND(1)** can have any value for 0 to 1, and since its actual value should be random this gives a 70% chance that the computer will jump to line 300 of the program and skip past the routine which creates a monster and allows the player to fight with it.

```
210 IF RND(1)<0.7 THEN 300
```

The three lines of the monster fight routine set the monster flag **M** equal to **1** to indicate that a monster is present to the rest of the program. They also choose two random numbers between 1 and 10. These numbers are used to select both a random monster name, and a random adjective to describe that monster. Subroutines at lines 1500 and 1600 are used. These subroutines print out adjective number **X** and monster name number **X** respectively. Thus the values of the

variables **A** and **B** which hold the random numbers, are put into the variable **X** just before calling each subroutine.

```
220 M=1:A=INT(RND(1)*10+1):B=INT(RND(1)*
10+1)
230 PRINT " THERE IS A ";:X=A:GOSUB 1500
240 X=B:GOSUB 1600:PRINT " HERE"
```

```
1500 REM PRINT THE XTH ADJECTIVE
1510 Q=(X-1)*6+1
1520 PRINT A$(Q,Q+5);" ";
1530 RETURN
```

```
1600 REM PRINT THE XTH NAME
1610 Q=(X-1)*7+1
1620 PRINT N$(Q,Q+6);" ";
1630 RETURN
```

These two subroutines work because the start of the adjectives and monster names are located at known positions within the two string variables **A$** and **N$.**

Back to line 300 and the player is given a choice of several options. The player may opt to Fight, Parry, Move, or Run. The get command is used to read an ATASCII number into the variable **T** and the **CHR$** function is used to convert this number into a character, in the variable **T$**.

```
300 PRINT "WHAT DO YOU WANT TO DO"
310 PRINT :PRINT "FIGHT / PARRY / MOVE /
 RUN"
320 GET £1,T:T$=CHR$(T)
```

Line 330 checks that the player has typed a legal command letter. If a mistake has been made then the computer jumps back to line 300 and prints out the set of options again.

```
330 IF T$<>"F" AND T$<>"P" AND T$<>"M" A
ND T$<>"R" THEN PRINT " !!! PARDON !!!":
PRINT :GOTO 300
```

Lines 340 and 350 are used to trap the two silly responses which the player might have made. Either trying to walk away from a monster, or asking to fight when there is no monster nearby and hence nothing to fight. These two responses will result in a short message being printed, and the player is sent back to the list of options at line 300.

```
340 IF T$="M" AND M=1 THEN PRINT "YOU CA
N'T WALK AWAY FROM HERE !":GOTO 300
350 IF T$="F" AND M=0 THEN PRINT " !!! F
IGHT WHAT !!!":GOTO 300
```

If you choose to **RUN** away and there is no monster there then you will be moved to a random new location. Line 360 takes care of this, prints a short message, and jumps to the start of the new location routine at line 100.

```
360 IF T$="R" AND M=0 THEN PRINT "... GO
OD EXERCISE... I SUPPOSE...":GOTO 100
```

If you try to run away and there is a monster around then you have a sixty percent chance of being caught by the monster. Line 370 deals with this, and jumps to the fight routine at line 500 if you don't get away.

```
370 IF T$="R" AND RND(1)>0.6 THEN PRINT
"....YOU DIDN'T GET AWAY...":GOTO 500
380 IF T$="R" THEN 100
```

Line 380 allows the player to select the fight routine himself, rather than being forced into it.

```
385 IF T$="F" THEN PRINT "...BE ABSOLUTE
LY CLEAR...":PRINT "....THIS IS YOUR IDE
A":GOTO 500
```

and lines 390 and 400 deal with the parry option, which does not really come into use until the fight routine at line 500.

```
390 IF T$="P" AND M=0 THEN PRINT "...GOO
D PRACTISE...":PRINT ".......NOW ...":GO
TO 300
400 IF T$="P" THEN PRINT "...HANG ON SUN
SHINE":GOTO 500
```

Since the **F P** and **R** options have now been dealt with, and since values of **T$** other than **F,P,R** and **M** have already been rejected, at line 330, it is safe to assume that at line 410 the player wants to move.

Lines 410 to 450 ask the player for a direction to move in, and then read characters from the keyboard until a direction is typed which is a legal exit. This is found out by comparing the characters which the player types against each of the characters in **E$**.

The variable **F** is used as a flag variable by this part of the program to indicate whether or not a legal dlirection was selected.

```
410 PRINT :PRINT "WHICH DIRECTION ";E$
420 PRINT "? ";
430 GET £1,T:T$=CHR$(T):PRINT T$:F=0:FOR
 L=1 TO 6
440 IF E$(L,L)=T$ THEN F=1
450 NEXT L:IF F=0 THEN PRINT "....YOU CA
N'T GO THAT WAY":GOTO 410
```

The program then jumps back to the start of the new room routine at line 100. If either Up or Down movement is chosen then the LeVel variable **LV** is altered and special messages are printed by lines 460 or 470. Line 480 jumps back for a new room.

```
460 IF T$="U" THEN LV=LV-1:PRINT "YOU'LL
 GET OUT OF HERE YET..."
470 IF T$="D" THEN LV=LV+1:PRINT "DEEPER
 AND DEEPER DELVING IS":PRINT "BAD FOR Y
OUR HEALTH"
480 GOTO 100
```

Line 500 labels the start of the monster fight routine.

```
500 REM FIGHT MONSTER
```

Then the program chooses random strengths and magic powers for the monster between 1 and the maximum available for the monster in question. This uses the variable **B** which contains the monster number to select elements from the two arrays **ST** and **MG**.

```
505 S=INT(RND(1)*ST(B)+1):M=INT(RND(1)*M
G(B)+1)
```

A short menu of choices is then **PRINT**ed out at line 510. Line 520 reads from the keyboard and only allows legal commands through to line 530.

```
510 PRINT :PRINT "CAST SPELL / FIGHT / R
UN"
520 GET £1,T:T$=CHR$(T):IF T$<>"C" AND T
$<>"F" AND T$<>"R" THEN PRINT " ??? EH ?
??":GOTO 510
```

The player only has a small chance, about 15%, of being able to run away from a monster he is fighting. This is dealt with by line 530.

```
530 IF T$="R" AND RND(1)>0.15 THEN PRINT
 "YOU CAN'T GET AWAY":GOTO 600
```

If the player tries to run but can't get away then the monster is allowed to attack the player. Hence line 530 jumps to the monster attack routine at line 600. Otherwise if the execution of the program reaches line 540 and **T$** is set for the **Run** command then it is certain that the player got away and the program jumps to the start of the new location routine at line 100.

```
540 IF T$="R" THEN 100
```

Casting spells when your magic power is zero will not do the monster any damage, but it will be allowed to attack you. Line 550 jumps to the monster fight routine if you have no magic power left.

```
550 IF T$="C" AND MG<=0 THEN PRINT "YOU
HAVE USED UP ALL YOUR MAGIC":GOTO 600
```

If you choose to fight the monster then you have a 70% chance of hitting. If you miss, i.e. if a random number turns out to be greater than 0.7, then line 560 will spot this and inform you that you missed. It will then jump to the monster attacks player routine at line 600. If line 560 doesn't decide that you missed then you have, presumably, hit the monster. Line 570 deals with this and reduces the value of the monsters strength which is in the variable **S** by a random amount between 1 and the players strength.

```
560 IF T$="F" AND RND(1)>0.7 THEN PRINT
"YOU MISSED":GOTO 600
570 IF T$="F" THEN S=S-INT(RND(1)*ST+1):
PRINT "A HIT ....HOW VIOLENT":GOTO 600
```

At line 580 it can be assumed that the player has successfully cast a magic spell, because all other options have already been dealt with. Therefore line 580 damages the monster, by reducing the variable **S**, and reduces the players magic power by the same amount, in the variable **T**. It also prints a suitable message.

```
580 T=INT(RND(1)*MG+1):S=S-T:MG=MG-T:PRI
NT "..POW..ZAP...WHAT A PRETTY FLASH":GO
TO 600
```

Line 600 is another comment or **REMark** line and labels the start of the monster attack routine.

```
600 REM MONSTER ATTACK
```

The program then checks to see if the monster has already been killed by the player, if it has then it is not allowed to attack.

```
610 IF S<=0 THEN PRINT "..YOU KILLED THE
 ";:X=A:GOSUB 1500:X=B:GOSUB 1600:PRINT
:GOTO 650
```

A random number between 1 and 100 is chosen, at the start of line 620. This is compared with the monsters **FI**ght number which is a measure of the probability that the monster will **FI**ght instead of casting a spell.

The rest of the monster attack routine performs the same task as the player attack routine did, but reduces the player's health according to the monsters strength and magic power.

```
620 T=INT(RND(1)*100):IF T>FI(B) AND M<>
0 THEN PRINT "..WHAM..A SPELL..":GOTO 64
0
630 PRINT "THE FIEND TRIES TO HIT YOU":I
F RND(1)>0.7 THEN PRINT "BUT MISSES...":
GOTO 648
633 PRINT "AND SUCCEEDS"
636 HL=HL-INT(RND(1)*S+1):PRINT "OH DEAR
! MORE DAMAGE":GOTO 645
640 T=INT(RND(1)*M+1):M=M-T:HL=HL-T
```

The program must then check to see if the player is still alive. This is done at line 645, and a message is printed at line 648 if he is alive and well. The routine then jumps back to the start of the player attacks routine.

```
645 IF HL<=0 THEN 800
648 PRINT "STILL ALIVE, BETTER THAN I TH
OUGHT"
649 GOTO 500
```

The routine from line 650 to 695 is executed when the player kills the monster. It increases the players score by giving him a random amount of gold based on the ferocity of the monster. It ends by jumping back to the section of the new room routine which tells the player where the exits are.

```
650 REM CREATURE IS DEAD
660 PRINT "WELL DONE BRAVE KNIGHT..."
670 T=INT(RND(1)*(MG(B)*2+ST(B)))*5
680 PRINT "HAVE ";T;" PIECES OF GOLD"
690 SC=SC+T:M=0
695 GOTO 180
```

If the player is killed during the monster attacks routine then the computer will jump to line 800. This part of the program prints a suitable message to indicate that the player has died.

```
800 REM PLAYER IS DEAD
810 PRINT :PRINT
820 PRINT "THE WHOLE COURT WILL MOURN"
830 PRINT "THE PASSING OF A BRAVE WARRIO
R"
840 PRINT "KNOWN FOR HIS VALOUR AND STRO
NG WILL"
850 PRINT "   (I.E.    YOU'RE DEAD MATE)
    "
860 FOR L=1 TO 600:NEXT L
870 PRINT "BY THE WAY YOU SCORED ";SC
890 END
```

The similar routine from line 900 to 960 is executed if the player gets out of the maze. This is noticed by the **IF** command at line 102. When the variable **LV** reaches zero.

```
900 REM PLAYER GOT OUT
905 GRAPHICS 0:POKE 752,1:PRINT
907 POSITION 0,12:PRINT £6;"      YOU ES
CAPED"
910 FOR L=1 TO 255:POKE 710,L:POKE 712,(
255-L)
920 NEXT L
930 PRINT "YOUR SCORE IS :"
940 PRINT "      ";SC
948 PRINT "PLUS A SMALL PSYCHOLOGICAL AD
VANTAGE"
949 FOR L=1 TO 500:NEXT L:PRINT :PRINT "
...YOU'RE ALIVE"
950 GET £1,A
960 END
```

That concludes the program Wizards and Warlocks. When you have typed it all in check it carefully against the complete listing which follows and then **RUN** it. Make sure that you can understand how each portion of the program is working. Notice how the program neatly divides into sections, each of which performs a particular task. It is often a good idea to write large programs in smaller sections which can be **RUN** independently of the main program and de-bugged separately.

```
10 REM WIZARDS AND WARLOCKS
20 DIM N$(80),MG(10),ST(10),FI(10),A$(70
)
25 DIM T$(10),E$(6)
30 GOSUB 1000
40 GRAPHICS 0
50 OPEN £1,4,0,"K:"
60 LV=6:REM STUCK ON SIXTH LEVEL
70 SC=0
80 ST=10:HL=ST+5:MG=2
99 REM NEW LOCATION
100 E$(1,6)="      "
101 SC=SC+1
102 IF LV=0 THEN 900
105 M=0
107 MG=MG+1
109 IF HL<ST+5 THEN HL=HL+1
110 IF RND(1)>0.7 THEN E$(1)="N     "
120 IF RND(1)>0.7 THEN E$(2)="E    "
130 IF RND(1)>0.7 THEN E$(3)="S   "
140 IF RND(1)>0.7 THEN E$(4)="W  "
150 IF RND(1)>0.95 THEN E$(5)="U "
160 IF RND(1)>0.95 THEN E$(6)="D"
165 IF E$="      " THEN 110
170 PRINT "YOU ARE IN A NEW ROOM"
180 PRINT :PRINT "THERE ARE EXITS :"
190 PRINT "     ";E$
200 PRINT
210 IF RND(1)<0.7 THEN 300
220 M=1:A=INT(RND(1)*10+1):B=INT(RND(1)*
10+1)
230 PRINT " THERE IS A ";:X=A:GOSUB 1500
240 X=B:GOSUB 1600:PRINT " HERE"
300 PRINT "WHAT DO YOU WANT TO DO"
310 PRINT :PRINT "FIGHT / PARRY / MOVE /
 RUN"
320 GET £1,T:T$=CHR$(T)
330 IF T$<>"F" AND T$<>"P" AND T$<>"M" A
ND T$<>"R" THEN PRINT " !!! PARDON !!!":
PRINT :GOTO 300
```

```
340 IF T$="M" AND M=1 THEN PRINT "YOU CA
N'T WALK AWAY FROM HERE !":GOTO 300
350 IF T$="F" AND M=0 THEN PRINT " !!! F
IGHT WHAT !!!":GOTO 300
360 IF T$="R" AND M=0 THEN PRINT "... GO
OD EXERCISE... I SUPPOSE...":GOTO 100
370 IF T$="R" AND RND(1)>0.6 THEN PRINT
"....YOU DIDN'T GET AWAY...":GOTO 500
380 IF T$="R" THEN 100
385 IF T$="F" THEN PRINT "...BE ABSOLUTE
LY CLEAR...":PRINT "....THIS IS YOUR IDE
A":GOTO 500
390 IF T$="P" AND M=0 THEN PRINT "...GOO
D PRACTISE...":PRINT ".......NOW ...":GO
TO 300
400 IF T$="P" THEN PRINT "...HANG ON SUN
SHINE":GOTO 500
410 PRINT :PRINT "WHICH DIRECTION ";E$
420 PRINT "? ";
430 GET £1,T:T$=CHR$(T):PRINT T$:F=0:FOR
 L=1 TO 6
440 IF E$(L,L)=T$ THEN F=1
450 NEXT L:IF F=0 THEN PRINT "....YOU CA
N'T GO THAT WAY":GOTO 410
460 IF T$="U" THEN LV=LV-1:PRINT "YOU'LL
 GET OUT OF HERE YET..."
470 IF T$="D" THEN LV=LV+1:PRINT "DEEPER
 AND DEEPER DELVING IS":PRINT "BAD FOR Y
OUR HEALTH"
480 GOTO 100
500 REM FIGHT MONSTER
505 S=INT(RND(1)*ST(B)+1):M=INT(RND(1)*M
G(B)+1)
510 PRINT :PRINT "CAST SPELL / FIGHT / R
UN"
520 GET £1,T:T$=CHR$(T):IF T$<>"C" AND T
$<>"F" AND T$<>"R" THEN PRINT " ??? EH ?
??":GOTO 510
530 IF T$="R" AND RND(1)>0.15 THEN PRINT
 "YOU CAN'T GET AWAY":GOTO 600
540 IF T$="R" THEN 100
```

```
550 IF T$="C" AND MG<=0 THEN PRINT "YOU
HAVE USED UP ALL YOUR MAGIC":GOTO 600
560 IF T$="F" AND RND(1)>0.7 THEN PRINT
"YOU MISSED":GOTO 600
570 IF T$="F" THEN S=S-INT(RND(1)*ST+1):
PRINT "A HIT ....HOW VIOLENT":GOTO 600
580 T=INT(RND(1)*MG+1):S=S-T:MG=MG-T:PRI
NT "..POW..ZAP...WHAT A PRETTY FLASH":GO
TO 600
600 REM MONSTER ATTACK
610 IF S<=0 THEN PRINT "..YOU KILLED THE
 ";:X=A:GOSUB 1500:X=B:GOSUB 1600:PRINT
:GOTO 650
620 T=INT(RND(1)*100):IF T>FI(B) AND M<>
0 THEN PRINT "..WHAM..A SPELL..":GOTO 64
0
630 PRINT "THE FIEND TRIES TO HIT YOU":I
F RND(1)>0.7 THEN PRINT "BUT MISSES...":
GOTO 648
633 PRINT "AND SUCCEEDS"
636 HL=HL-INT(RND(1)*S+1):PRINT "OH DEAR
! MORE DAMAGE":GOTO 645
640 T=INT(RND(1)*M+1):M=M-T:HL=HL-T
645 IF HL<=0 THEN 800
648 PRINT "STILL ALIVE, BETTER THAN I TH
OUGHT"
649 GOTO 500
650 REM CREATURE IS DEAD
660 PRINT "WELL DONE BRAVE KNIGHT..."
670 T=INT(RND(1)*(MG(B)*2+ST(B)))*5
680 PRINT "HAVE ";T;" PIECES OF GOLD"
690 SC=SC+T:M=0
695 GOTO 180
800 REM PLAYER IS DEAD
810 PRINT :PRINT
820 PRINT "THE WHOLE COURT WILL MOURN"
830 PRINT "THE PASSING OF A BRAVE WARRIO
R"
840 PRINT "KNOWN FOR HIS VALOUR AND STRO
NG WILL"
850 PRINT "  (I.E.   YOU'RE DEAD MATE)
    "
```

```
860 FOR L=1 TO 600:NEXT L
870 PRINT "BY THE WAY YOU SCORED ";SC
890 END
900 REM PLAYER GOT OUT
905 GRAPHICS 0:POKE 752,1:PRINT
907 POSITION 0,12:PRINT £6;"      YOU ES
CAPED"
910 FOR L=1 TO 255:POKE 710,L:POKE 712,(
255-L)
920 NEXT L
930 PRINT "YOUR SCORE IS :"
940 PRINT "      ";SC
948 PRINT "PLUS A SMALL PSYCHOLOGICAL AD
VANTAGE"
949 FOR L=1 TO 500:NEXT L:PRINT :PRINT "
...YOU'RE ALIVE"
950 GET £1,A
960 END
1000 REM READ IN CHARACTERISTICS
1001 REM OF MONSTERS
1010 RESTORE :N$=""
1020 FOR L=1 TO 10
1030 READ T$:N$(LEN(N$)+1)=T$
1040 NEXT L
1050 A$=""
1060 FOR L=1 TO 10
1070 READ T$:A$(LEN(A$)+1)=T$
1080 NEXT L
1100 FOR L=1 TO 10
1110 READ A,B,C
1120 MG(L)=A:ST(L)=B:FI(L)=C
1130 NEXT L
1199 RETURN
1500 REM PRINT THE XTH ADJECTIVE
1510 Q=(X-1)*6+1
1520 PRINT A$(Q,Q+5);" ";
1530 RETURN
1600 REM PRINT THE XTH NAME
1610 Q=(X-1)*7+1
1620 PRINT N$(Q,Q+6);" ";
1630 RETURN
```

```
9000 DATA ELF     ,DWARF   ,TROLL   ,WIZARD
 ,TOAD    ,WARLOCK
9010 DATA ZOMBIE ,BEETLE ,ORC     ,CHIMER
A
9100 DATA CRUEL ,GREEDY,FOUL   ,WICKED,SL
IMY ,SMELLY,CLEVER
9110 DATA LARGE ,SLY    ,HONEST
9200 DATA 5,5,10
9210 DATA 1,8,70
9220 DATA 0,15,100
9230 DATA 20,7,5
9240 DATA 0,20,100
9250 DATA 0,3,100
9260 DATA 3,10,65
9270 DATA 0,5,100
9280 DATA 2,8,50
9290 DATA 10,40,50
```

7

Saving and Growing

One of the drawbacks with home computers is that they forget programs, variables and everything else which they have been doing when the power is turned off. In order to avoid having to type in your programs every time you want to use them, you must have some method of making permanent copies of programs. Most home computers can use either tape cassettes or floppy disks to store programs on. The Atari is no exception.

You can purchase an Atari data recorder, which is basically a specialised tape recorder, and you will then be able to store your programs on normal tape cassettes. You can also use a disk drive and store programs on floppy disks. However disk drives are quite expensive and so you will probably use cassette storage for some time before you progress to using disk storage.

The **ATARI 1010 PROGRAM RECORDER** is the name of the data recorder which is designed for use with the **600XL.** The program recorder connects to the computer via a large wedge shaped plug which fits into the **PERIPHERAL** socket. The peripheral socket is located on the rear edge of your computer behind the **RESET** key. The program recorder also requires to be connected to it's own power supply. You will probably need to purchase a three or fourway plugboard since you now need three sources of mains power. For the computer, television, and program recorder.

If you want to load and run commercially produced programs, and let's face it a lot of people spend a lot of time playing arcade quality games on their Atari computers, then you should follow the instructions provided with the program on how to load it into the computers memory from tape. If no instructions are given then try the following.

1) Type **CLOAD** on your computer and press **RETURN**.

2) You will now be able to rewind the tape in the data recorder to the beginning of the side.

3) Press **RETURN** on the computer again.

4) Press the **PLAY** button on the data recorder.

The program should now load, it will probably run automatically, but if it doesn't then type **RUN** and press **RETURN**.

Eventually you will write computer programs which you will want to save on tape before turning off your computer. To save programs on tape we use the **CSAVE** command. When you want to save a program on cassette put a blank tape in the data recorder and position the tape at the start of the side on which you want to save your program.

1) Type **CSAVE** and press return. This will allow you to position the tape using **REWIND** and **ADVANCE**.

2) When the tape is in the correct position press the **RECORD** and **PLAY** keys on the program recorder together.

3) Now press **RETURN** once more.

The program will be saved to cassette. You can now use the **CLOAD** command to recover your program at a later date.

It is a good idea to save your programs on short length 'computer' tapes. **C12** and **C15** are the most common sizes. Label each cassette with the name and purpose of the program. Also write down any notes which would prove helpful to anyone else who might use the program.

From time to time you should clean your program recorder's read and erase heads. You can do this with a cleaning cassette, identical to one which you might use to clean a HI-FI tape recorder.

Remember to store your tapes away from any electrical unit which contains a magnet as this will erase your programs.

As you become more and more familiar with your computer you will discover many tasks which it could be used for if you only had extra bits of equipment. Printers, joysticks and data recorders are among the most common 'add-ons' which people get for their computers.

With a printer attached to your computer you can make permanent copies of the results of your programs. You can use the computer as a wordprocessor to write your letters faster and more efficiently. There are three different printers which are designed for use with your computer.

1) The Atari 1020 is a printer which can print in up to four colours. The colours can also be drawn close together to produce shading effects. This printer can be very useful when you want to make charts and diagrams in colour. It can of course be used for listing programs.

2) The Atari 1025 printer can print on wider paper than the colour printer. The quality of the print is also more suited for writing letters. However only one colour, usually black, is available for output. This printer produces it's output as sets of closely grouped dots. Hence the printer is known as a

DOT-MATRIX printer. Dot matrix printers are the most popular type of printers in use in homes because of their relatively low cost for the good quality printout they produce.

3) The Atari 1027 is a letter quality printer. This printer could be used for that all important letter to your bank manager. It also has the facility to produce more than one copy of a document. Naturally it costs a little more than the others.

Another common add on is a joystick. The Atari super joysticks are easy to use and will improve your aim at that crucial stage in many arcade games. You can also use the joysticks in your own programs. See the section on advanced programming. The old favourite games paddles can also be used with the 600XL.

Extra memory can be easily added to your computer. The 64K memory module will quadruple the amount of memory which your computer has for program, variable, and graphics storage. This will enable much larger programs to be written. This unit plugs into the **PARALLEL BUS** socket at the rear of your computer.

The major problem with using cassettes for storing programs from any home computer is that it can take a long time to save programs onto cassette, and just as long again to load them back into program memory. A few home computers have the facility to use Floppy disks to store programs. These have the advantage of being much faster than cassette. Your Atari computer can control an Atari 1050 disk drive. This unit will allow you to store programs, data and graphics on floppy disks.

As well as the aforementioned standard computer add-ons the 600XL can also handle other specialised devices such as touch tablets and trackballs. Trackballs are useful for some games, such as missile command, and for creative graphics programs. The touch tablet allows you to draw on your

computer's screen simply by moving a '**PEN**' over a special pad.

You will soon appreciate that the 600XL is a very versatile computer. With all these expansions and your imagination there is very little which your computer cannot do which other home computers can.

8

Programming Errors

There are three types of errors which can occur whilst programming your computer. **SYNTAX** errors, **EXECUTION** errors and **LOGICAL** errors.

SYNTAX errors are errors in the **SYNTAX** of your program. That is you have spelt commands or functions wrongly or used the wrong number, or the wrong type, or parameters or arguments. **SYNTAX** errors will nearly always be spotted by the computer when you press <**RETURN**> after typing in the offending program line.

An example of a syntax error might be:

```
10 PIRNT "HELLO"
```

to which the computer will respond

```
10 ERROR- PIRNT"HELLO"
```

the first quote mark will be highlighted by being in inverse video, that is black marks on a white background. The way to get rid of this error is to correct the spelling of the **PRINT** command. Simply retype the line. You could of course use the cursor control keys to copy most of the original line because the amount of change required is small.

EXECUTION errors only occur while the program is running, and are reported to you by the computer by the **ERROR** report. The general form of an **ERROR** report is :

ERROR 12 AT LINE 10

which would occur if **LINE 10** was executed and contained a **GOTO** or **GOSUB** to a program line which didn't exist.

There are many different error codes which may be reported as **EXECUTION** errors. A list giving possible causes follows.

ERROR 2 means that your program and variables together are using more memory than the computer has got. You can check on this with the **FRE(0)** function (see the advanced programming chapter). You must reduce the size of your program or the number of variables. One common cause is the practice of **DIMensioning** strings and arrays much larger than they need be **DIMensioned**. You can reduce the amount of memory which your program uses by putting more than one command on each program line, separating them with colons.

ERROR 3 means that you have used a number or character for a command or function which cannot be used. For instance trying to execute **PRINT CHR$(-1)** will give error 3 because the value in the brackets is not in the **ATASCII** code. Similarly attempts to dimension strings or arrays with negative sizes will give error 3.

ERROR 4 means that too many variables have been used. The computers variable memory has reached its limit. You must cope with fewer variables.

ERROR 5 This error occurs when you attempt to use a string variable which has either not been **DIMensioned** large enough, or has not been dimensioned at all. Check that you have correctly dimensioned the string concerned. If the string is dimensioned correctly then check that the program does not try to access beyond the end of the string. Any attempt to access the zeroth character of a string, for example with, **A$(0)="g"**, will automatically give error 5.

ERROR 6 This error occurs when a **READ** statement is executed and there is no more **DATA** available. Check that all the **DATA** statements which should be present are there. Also check that the part of the program which **READs** the **DATA** isn't trying to read too much data. For instance if you are **READing** data within a **FOR/NEXT** loop then the loop may be being executed too many times.

ERROR 7 this error occurs if your program makes reference to a line number greater than 32767. Line 32767 is the highest line number which you can use in Atari basic. The error will occur, for example, if you try to execute

GOTO 50231

or

RESTORE 40287

this error only occurs in immediate mode that is when you are typing commands without line numbers. The same type of mistake in a program will give a syntax error, **ERROR 17**.

ERROR 8 this error occurs if the user types a response to an **INPUT** command which is not consistent with the variable into which the computer is asked to put the response. For example trying to **INPUT** a string of letters into a numeric variable will give this error. This error will also occur if you try to **READ** a string into a numeric variable.

ERROR 9 this error will occur if you try to **DIMension** a string of zero length. It will also occur if you try to create a string longer than 255 characters. This error is also used to indicate when you try to **DIMension** the same variable twice without using the **CLR** command.

ERROR 10 this error occurs when the argument stack which the computer uses to process mathematical expressions gets full. Try to split up your calculations into smaller parts.

ERROR 11 this error occurs when the temporary variables which the computer uses to hold the results, and partial results, of calculations overflows or underflows. That is the number held on it gets too big or too small to be accurate. You must redesign the calculation.

ERROR 12 this error occurs when you attempt to **GOTO** or **GOSUB** a line which doesn't exist. Check that you haven't made a mistake typing the line at which the error has occured. Also check that you haven't missed out the line which the computer is trying to jump to.

ERROR 13 NEXT without **FOR** error. This error occurs when a **NEXT** command makes reference to a variable which wasn't the variable corresponding to the last **FOR** command encountered. This usually means that you have made a typing mistake at either the **FOR** command or **NEXT** command concerned. Be wary that each time a **NEXT** command is met by the computer running your program, the variable referenced is the same as that in the last **FOR** command.

ERROR 14 occurs when a program line is too long. If you exceed the maximum length then you will get this error.

ERROR 15 doesn't occur very often. It is caused when you have a legal **FOR/NEXT** or **GOSUB/RETURN** pair of commands but the **FOR** line or **GOSUB** line is somehow deleted before the loop or subroutine is finished. When the terminating **NEXT** or **RETURN** command is reached this error is given.

ERROR 16 this error occurs when a **RETURN** command is encountered without a **GOSUB** command having occurred.

ERROR 17 this error code is used to indicate **SYNTAX** errors which weren't spotted whilst the program was being typed in. Check that the offending program line has been typed in correctly.

ERROR 18 this error doesn't occur very often. It is caused when you try to **INPUT** invalid characters into String variables.

ERROR 19 this error is caused when you try to load a program into your computer which is bigger than the available free memory. This may be caused if you try to load a program written on an **ATARI 800** or **800XL.**

ERROR 21 this error can occur when loading a program from tape. Rewind the tape and try again. If the error persists then you are unlikely to be able to **LOAD** that program. For this reason it is a good idea to keep several copies of important programs on different tapes.

ERROR 140,142 and **143** can also occur when loading from tape. Rewind and try again. (see **ERROR 21**).

LOGICAL errors are usually the most difficult to trace to their source and remove. **LOGICAL** are errors in the logic of the program which you have written. They often make themselves apparent when you notice that your program isn't doing what it is supposed to do.

Logical errors are best found by a careful examination of the part of the program which isn't behaving properly. First of all decide exactly which part of the program isn't working. Then make a note of all the other parts of the program which use the same variables as this section.

You must now play the part of the computer, very carefully perform the task of the computer. Execute statements in order, writing down the contents of variables, strings and arrays as they change. Try to find out which part of the program is not doing what you thought it was. This process of pretending to be the computer is known as '**dry-running**'.

You will almost certainly discover your error while dry-running the section of the program which isn't behaving. If not then try expanding the amount of program which you are examining, until you have taken in the whole program. This will find your error, but it may take a long time.

This process of hunting for the cause of errors is known as **de-bugging**. Most computer programmers spend a lot of their time engaged in de-bugging. The difficulty of de-bugging increases as the size of the program increases. This is one reason why many programmers like to write their programs in sections, or subroutines. In basic these subroutines are called with the **GOSUB** command.

If the program is written as a set of smallish subroutines which can be completely de-bugged and tested by themselves then the debugging of the whole program will be made far easier, since you will then only have to de-bug small amounts of program at any one time.

One of the best ways to learn to remove all types of errors from your programs is to write lots of small programs which perform fairly simple tasks, and to gradually increase the complexity of the programs as your experience increases.

Try to get a friend who has more computing experience than yourself to help you de-bug a program or subroutine if you get really stuck. You will probably learn a lot from watching the way they attack the problem. Eventually you will become an efficient de-bugger but this takes time and practice, so don't try to write that brilliant arcade game you had planned first time, it will be very hard to debug. Start with small bugs and work your way up to really complicated ones.

GLOSSARY

OF ATARI BASIC RESERVED WORDS

Reserved words are special words which are used to control the computer in **ATARI BASIC**. Reserved words cannot be used as variable names because the computer recognises them as special words. They have been split into two alphabetical lists here, the second list contains more advanced reserved words which you will probably not need until you have been using the computer for some time. However their use is explained for you to use when you have mastered the simpler aspects of **ATARI BASIC**.

The reserved words are divided into two classes. Commands, which usually cause things to happen. Examples of Commands would be **PRINT** and **SOUND**; and Functions, which are used either as assignment commands, with the **LET** command or = sign, or as part of the condition of an **IF** statement. Examples of Functions would be **SIN()** and **ASC()**. Functions always have the Numbers, Numeric variables, or Strings associated with them and enclosed in brackets immediately after the Function name.

? this is an abbreviation for the **PRINT** command. It does exactly the same as the **PRINT** command.

ABS() this function returns the **ABS**olute part of the number in the brackets. That is, if the number in brackets is negative the minus sign will be removed. E.G.; **ABS(–5.23)** is equal to **5.23**, otherwise no change will take place.

AND this command is used to separate two conditions in an **IF** command. The overall condition will only be true if both the first **AND** second conditions are true.

ASC() this function returns the ATASCII code number for the letter in the brackets. String variables can also be put in the brackets, in which case the ATASCII code of the first character of the string will be returned.

ATN() this function returns the ATASCII code number for the letter in the brackets. String variables can also be put in the brackets, in which case the ATASCII code of the first character of the string will be returned.

ATN() this function returns the **A**rc**T**a**N**gent of the number in the brackets. Numeric variables can also be used in the brackets, as they can with any of the number based functions. The **DEG** or **RAD** commands decide whether the number returned will be in **DEG**rees or **RAD**ians.

CHR$() this function returns a one character long String. That is, a String with a **LEN**gth of 1. The character returned will be the character whose ATASCII code number is in the brackets.

CLOAD this command asks the computer to load a program from cassette. The first program which the computer finds on the cassette will be loaded.

CLOG() this function returns the **C**ommon **LOG**arithm (that is a **LOG**arithm base 10) of the number in brackets.

CLR this command **CL**ea**R**s any knowledge of Numeric, String or Array variables which the computer has. It has the same effect upon Variable memory as the command **NEW** has upon both Variable and Program memory.

COLOR this command tells the computer which colour to draw in. Note the American spelling of the word **COLOR**. One number follows the command. This number must be an integer between Ø and 15.

CONT this command **CONT**inues **RUN**ning the program in the computer's memory from the point which was reached when a **STOP** or **END** command was found in the program. It can also be used to carry on after the **RUN**ning of a program is interrupted with the **BREAK** or **RESET** keys. This will only work with **BASIC** programs.

COS() this function returns the **COS**ine of the angle in the brackets. The angle may be either **DEG**rees or **RAD**ians depending upon which has been selected with the **DEG** or **RAD** commands. The computer will assume that you want to use **RAD**ians unless you include a **DEG** command somewhere in your program.

CSAVE this command **SAVES** the contents of program memory to cassette tape. Remember to note whereabouts on the tape you record different programs so that you will know where to find them.

DATA this command is followed by a list of Numeric or String **DATA**. The data can be read with the **READ** command. The computer has an internal pointer which it uses to keep track of where the next item of **DATA** to be **READ** is to come from. The contents of this pointer can be set to point at the start of any line with the **RESTORE** command.

DEG this command tells the computer that you want all subsequent trigonometric functions, that is (**ATN, SIN** and **COS**), to work in **DEG**rees.

DIM this command is used to set the maximum **DIM**ensions of String variables and arrays. Only arrays of Numeric variables can be **DIM**ensioned. Arrays must have either one or two **DIM**ensions. E.G. **DIM FRED(4,7)** will produce an array called **FRED** which has 28 elements.

FRED(1,1).........FRED(4,1)

```
  ..                 ..
  ..                 ..
  ..                 ..
  ..                 ..
```

FRED(1,7).........FRED(4,7)

Strings must be **DIM**ensioned before they can be used.

DRAWTO this command will cause the computer to **DRAW** a line **TO** the point whose x and y co-ordinates are given after the command from the last point **DRAW**n **TO** or **PLOT**ted. Drawing will be in the colour most recently selected with the **COLOR** command.

END this command stops a program **RUN**ning. It also closes all open files (see the advanced section) and turns off the sound. It is used to mark the **END** of a program.

EXP() **this function returns the number e (approximately equal to 2.718) raised to the power of the number in the brackets. The accuracy of EXP** is only guaranteed to 6 significant digits. This function is the inverse of the **LOG**arithm function.

FOR this command, together with the **NEXT** command, provides a means of setting up loops in your programs. The **STEP** command can also be used to vary the **STEP** which is used to increment the loop variable each time the **NEXT** command is encountered. E.g.

FOR LOOP=A TO B STEP C

(some instructions)

NEXT LOOP

will execute (some instructions) with values of **LOOP** ranging from **A** to **B**. The value of **C** will be added to **LOOP** each time the **NEXT LOOP** command is reached. The loop will continue to be executed until **LOOP** becomes equal to or passes the value of **B**.

FRE(Ø) this command returns the amount of **FREe** memory available for **BASIC** at any moment. Note that the value in the brackets does not affect the number returned. Handy if you are writing long programs and want to know how much memory is left to be used.

GOSUB this command causes the computer to continue running the program from the line number specified. The number specified can be in the form of a mathematical expression. E.g.

GOTO 10+A*3

would be allowed but it would be hard to find mistakes in a program written with instructions like this.

GRAPHICS this command is used to select a graphics mode. Legal modes range from Ø to 15. If you add 16 onto the mode number then the whole screen will be devoted to graphics. But otherwise a small portion at the bottom may be reserved for text. If you add 32 onto the mode number then the new graphics mode will be selected but the screen will not be cleared.

TABLE OF MODES AND SCREEN FORMAT

SCREEN FORMAT

Graphics Mode	Mode Type	Columns	Rows — Split Screen	Rows — Full Screen	Number of Colors
0	TEXT	40	—	24	1-1/2
1	TEXT	20	20	24	5
2	TEXT	20	10	12	5
3	GRAPHICS	40	20	24	4
4	GRAPHICS	80	40	48	2
5	GRAPHICS	80	40	48	4
6	GRAPHICS	160	80	96	2
7	GRAPHICS	160	80	96	4
8	GRAPHICS	320	160	192	1-1/2
9	GRAPHICS	80	—	192	1
10	GRAPHICS	80	—	192	9
11	GRAPHICS	80	—	192	16
12	GRAPHICS	40	20	24	5
13	GRAPHICS	40	10	12	5
14	GRAPHICS	160	160	192	2
15	GRAPHICS	160	160	192	4

IF this command is used with the **THEN** command to allow the program to make decisions. E.g.

IF A=1 OR S$="FRED" THEN GOSUB 5000

will execute a subroutine starting at line 5000 **IF** the Numeric variable **A** is equal to **1 OR IF** the String variable **S$** is equal to the String **FRED**.

INPUT this command is used to **INPUT** from the keyboard to Numeric or String variables. Question marks will be printed on the screen until the required number of pieces of information have been typed. The **RETURN** key or commas must be used to separate different bits of information required by the same **INPUT** command.

INT() this function returns the **INT**eger part of the number or mathematical expression in the brackets.

LEN this function returns the **LEN**gth of the String or String expression in the brackets.

LET this command can be used in front of assignment commands. E.g.

A=4:F$"A WORD OR TWO"

is equivalent to

LET A=4:LET f$="A WORD OR TWO"

LIST this command **LIST**s out the whole, or part, of the program memory. If no numbers are given then the whole of the program memory will be **LIST**ed out. If one number is given then that line alone will be **LIST**ed. If two numbers are given then the portion of program memory between and including those two lone numbers will be **LIST**ed.

LOCATE this command is used to set a Numeric variable equal to the character or colour of a particular point on the screen. In text modes (0-2) the ATASCII codes will be transferred into the variable given. In graphics modes the colour of the specified point on the screen will be transferred. The form of the command is

LOCATE, X, Y, VALUE

which will transfer the ATASCII code, or colour, of position **X,Y** on the screen. N.b. This command also sets the last point visited memory, and so it may mess up your drawings if you aren't expecting this.

LOG() this function returns the natural logarithm (base e) of the number in the brackets. This is the inverse of the **EXP** function.

NEW this command clears the program memory and variable memory and instructs the computer that you want to write a **NEW** program.

NEXT together with the **FOR** and **STEP** commands this command is used to set up loops in BASIC programs.

ON GOSUB this command is used to choose from a set of subroutines. E.g.

ON A GOSUB 100,200,300

will call a subroutine at line 100 of the variable **A** is equal to **1**. If **A=2** then the subroutine at line 200 will be called. If **A=3** then the subroutine at line 300 will be used. It is important to make sure that you have sufficient subroutines to cater for all possible values of the variable or expression which you use to select a subroutine, otherwsie an error will be caused.

ON GOTO this command is similar to **ON GOSUB** but instead of calling a subroutine at a particular line number the computer will **GOTO** that line.

OR this command is similar to the **AND** command. It is used to separate two conditions within the **IF** statement. The overall condition will be true if either the first **OR** the second **OR** both conditions are true.

PLOT this command only works in graphics modes (3–15). It causes a point to be **PLOT**ted on the screen at the co-ordinates given in the current colour, which is selected with the **COLOR** command.

POSITION this command is used to tell the computer whereabouts on the screen the next bit of information to be **PRINT**ed is to go. It is only applicable to the three text modes (0 - 2). In all three of these modes the **POSITION** command is used to set the **X-Y POSITION** at which to **PRINT**. In mode 0 the **PRINT** command can then be used to **PRINT** at the selected point on the screen. However in modes 1 and 2 the extended command **PRINT 6** must be used. (This means **PRINT to channel number 6**; which is automatically opened

by the computer and which will **PRINT** to the screen. For a description of channels see the description of the **PRINT** command and **OPEN** command in the second part of this glossary).

PRINT this command is used to **PRINT** information on the screen. If text is included after the command, in double quotes, then it will be **PRINT**ed out. The contents of string and numeric variables can also be **PRINT**ed. String and arithmetic expressions can also be included in after the **PRINT** command. Commas can be used to organise the **PRINT**ed data into neat columns, and semi-colons can be used to keep the separate parts of data close together when **PRINT**ed out.

RAD this command is used to select **RAD**ians as the type of angular measurement to be used with the trigonometric functions.

READ this command is used to **READ** String or numeric **DATA**, from **DATA** statements, into String or Numeric variables. N.b. you will get an error if you attempt to read String data into numeric variables. The **RESTORE** command can be used to alter, or restart, the sequence in which the data is read.

REM this command is used to include **REM**arks about a program within the program memory. It is useful to be able to leave comments within a program so that you will remember what particular parts of a program do. It is good practice to label subroutines and to describe their function on the line before. Everything following the **REM** command on a line is ignored, including colons, so you cannot have a multi-statement line with a **REM** command at the start.

RESTORE this command is used to set the pointer to the next item of **DATA** to be **READ** to the start of any line in the program. If no number follows the command then the pointer

will be set to the first item of **DATA** in the program. If a number is given then the pointer will be set to the first item of **DATA** within the program on, of after, the line requested.

RETURN this command at the end of a subroutine will cause the computer to go back to the point in the program immediately after the **GOSUB** command which called the subroutine.

RND() this function returns a random number. The value in the brackets does not affect the number returned, however there must be a number in the brackets.

RUN this command causes the computer to execute the program in it's program memory starting at the first line, which will be the one with the lowest line number.

SETCOLOR this command is used to change the colours which correspond to the different **COLOR** numbers used for drawing. The **COLO**u**R**s can be set to any one of sixteen colours, each in 8 shades. Three numbers or Numeric variables or expressions follow the command. The first is the **COLOR** register number, usually one less than the **COLOR** you are trying to change. The second is the colour which you want to choose and the third is the shade you want to use.

SGN() this function returns the sign of the expression in the brackets.

SIN() this function returns the **SIN** of the angle in the brackets. The angle can be in either **RAD**ians or **DEG**rees, as long as the correct one has been selected with the relevant command.

SOUND this command is followed by four numbers separated by commas and makes a sound on one of four different sound channels depending upon the values of the numbers.

SQR() this function returns the **SQ**ua**R**e root of the expression in the brackets.

STEP this command is used as an option with the **FOR** command to allow loops which count in amounts other than 1.

STOP this command is used to stop a program from **RUN**ning. Unlike **END** it does not close any open files (see later on) or turn off the sound.

STR$ this function returns a string which is equivalent to the arithmetic expression in the brackets. It provides a way of transferring results of mathematical functions into strings where they can be printed with more control via the string manipulation commands.

THEN this command is used to separate the condition of the **IF** command from the command to be executed **IF** that condition is met.

VAL() this function returns a number which represents the string expression in the brackets. It is often used to allow versatile number input routines to be programmed and the results to be transferred into numeric variables.

ADVANCED FUNCTIONS AND COMMANDS

This section lists alphabetically the more advanced commands and functions which Atari Basic offers you. Many of these will not be needed until you have been programming for some time.

ADR() this function returns the address in memory at which the String or String variable begins. This can be used to allow Strings to be manipulated with the **PEEK** and **POKE** commands.

BYE this command exits from Basic and enters self-test mode. The ATARI 600XL has a program built into it which can be used to test some of the functions of the machine. The three functions which can be tested are:

1) The computer's memory

2) The computer's sound and graphics

3) The computer keyboard

The different tests are **SELECT**ed and **START**ed with the metallic keys to the right of the main keyboard.

CLOSE this command **CLOSES** a device after it has been opened for access with the **OPEN** command. The form of the command is

CLOSE #4

which will close the file open on channel 4. Legal channel numbers range from 1 to 7. Note that channel 6 is opened to the screen automatically by the computer. (See the **OPEN** command).

COM this command does exactly the same as the **DIM**ension command.

DOS this command selects the Disk Operating System and displays the **DOS** command menu. If used on a machine without disk drives then it will have the same effect as the **BYE** comamnd. That is self-test mode will be selected.

ENTER this command allows a program to be **ENTER**ed into the computer's program memory from an input device. The form of the command is

ENTER "C:"

which would **ENTER** the next program found on the cassette, or

ENTER "D:POEM"

which would **ENTER** a program called **POEM** from disk.

The program to be entered must have been saved with the **LIST"c:" or LIST"d:"** command. The advantage of **LIST**ing and **ENTER**ing programs rather than using **CSAVE** and **CLOAD** is that, if for some reason there is an error during **ENTER**ing the program, only one or two lines will probably be lost from the program.

If the program has been **CSAVE**d then the entire program would be lost.

The major disadvantage of using **ENTER** and **LIST** is that the amount of information which is saved on the tape or disk is greater, and so programs take longer to record in this way.

GET this command **GET**s a single character from a device which has previously **OPEN**ed. The form of the command is

GET #1,A

which will get one character from the device on channel 1 and put it's ATASCII code in the variable **A**. Note that only numerical variables can be got into, and that therefore the **CHR$()** function must be used to convert the characters which are got into String characters.

INPUT this command, as well as being used to **INPUT** to variables from the keyboard, can also be used to **INPUT** from a device which has been previously **OPEN**ed. The form of the command is

INPUT #1,A,BCD$

which would **INPUT from a device on channel 1 into the variables A** and **BCD$**. An error will be caused if the channel used has not been previously **OPEN**ed for input.

LIST this command is used to **LIST** a program, or section of a program, to an output device. It can be used to **LIST** a program to cassette, disk or printer. The commands to do these things would be:

LIST "C:"

LIST "D:"

LIST "P:"

in order to **LIST** a section of a program use commands such as

LIST "P:",10,50

which would **LIST** lines 10 to 50 to the printer.

LOAD this command is used to load a program from and input device. It is used to **LOAD** programs which have been **SAVE**d rather than **CSAVE**d or **LIST**ed. The command **LOAD"C:"** is equivalent to **CLOAD.**

LPRINT this command is used to **PRINT** to the printer. It is equivalent to the **PRINT** command except that the output is sent to the printer.

NOTE this command is used when disks are in use. It transfers the Sector and Byte location of the next byte to be read from or written to disk. E.g.;

NOTE # 1,A,B

will transfer the sector into **A** and the byte into **B**. The channel used must have previously **OPEN**ed for access to **DISK**.

OPEN this command is used to **OPEN** channels to devices. Legal channel numbers range from 1 to 7, though channel number 6 is automatically **OPEN**ed by the computer, and so must be **CLOSE**d before it can be **OPEN**ed for some other use. The form of the command is

OPEN # a,b,c,"C:"

The letter in quotes selects the device to which the channel is **OPEN**ed. The legal devices are:

C: CASSETTE

D: DISK

P: PRINTER

R: RS232 handler, ATARIA 850

S: SCREEN

K: KEYBOARD, characters typed are not automatically displayed on the screen

E: SCREEN EDITOR, S: and K: combined

D2: a second DISK drive

the first number is the channel number to which the device is to be assigned. The second number is the type of channel to be **OPEN**ed. Legal numbers are,

4 : Input channel only

8 : Output channel only

12 : Read and write

9 : Append to end of file

6 : Disk directory only

The third number is usually zero, it is used to control some specialised devices. For instance it was used to control sideways printing on old Atari printers.

PADDLE() this function returns the position of a game paddle. The number in brackets must be either Ø,1,2 or 3. The number returned varies from 1 to 228, increasing as the paddle is turned anti-clockwise.

PEEK() this function returns the contents of any byte of the computers memory. The number in brackets must be between Ø and 65535 which is the range of legal addresses in the Atari computer.

POINT this command is used to tell **DOS** (the Disk Operating System) where the next byte read from or written to disk is to come from. The form of the command is

POINT # 1,A,B

which would **POINT** the disk on channel number 1 at sector A, byte B.

POKE this command is used to alter the contents of any byte of the computer's memory.

POP this command can be used to exit a subroutine or a **FOR/NEXT** loop via the **GOTO** command. an error is caused if **GOTO** is used to exit either of these without first using the **POP** command to tell the computer that you intend to quit early.

PRINT this command has an extension to it's use as explained in the previous section. If a channel number is given before the data to be **PRINT**ed and that channel is **OPEN**ed for output, then the data to be printed will be transferred to the device on that channel. This can be used, for example, to **PRINT** data to tape for subsequent retrieval with the **INPUT** command.

PTRIG() this function returns a Ø if a specific paddle trigger is pressed. It returns a 1 if the trigger is pressed. It returns a 1 if the trigger is not pressed. The number in the brackets is the paddle number to investigate. This number must be either Ø,1,2 or 3.

PUT this command is used to **PUT** a single byte to a device on a given channel. The form of the command is

PUT # 1,24

which would **PUT** the byte 24 to a device on channel 1.

SAVE this command is used to save a program to any device. (See the **OPEN** command). **SAVE "C:"** is equivalent to **CSAVE**.

STATUS this command is used to transfer the **STATUS** of the device on a given channel into a numeric variable. A list of **STATUS** codes for different devices is given in the **ERROR** codes chapter. The form of the command is

STATUS # 1,A

which will set the variable A equal to the **STATUS** of the device on channel 1.

STICK() this function returns a number between Ø and 15 which corresponds to the position of one of the two joysticks. The number in brackets must be either Ø or 1, and selects which joystick is selected. The different positions and the values returned are shown below.

STRIG() this function returns a Ø if a specific joystick trigger button is pressed and a Ø if it isn't. The number in brackets, either Ø to 1, chooses the joystick.

TRAP this command sets the line number to which the computer should jump when an error occurs. It is often needed when programs which make use of devices are written. The number following the command is the line to which the computer will go if an error occurs.

USR() this command transfers control to a machine code routine starting at the address given in brackets. Additional numbers can be passed to the machine code routine by putting them after the address and separating them with commas.

INDEX

Page Nos.

GETTING STARTED with the ATARI 600XL

PETER GOODE

Aimed at first time users,
this book takes you from starting on the keyboard
and guides you, step by step,
until you become sufficiently expert
to write your own programs.

This *'essential'* book
will help you
use Atari Basic
understand graphics
design programs
file data on cassette.

Example programs are shown in the book
along with chapters on how to use
sound, colour and stretch your 600XL.

THE AUTHOR

PETER GOODE
is the *best selling* author
of several computer titles
including 'THE ATARI 600XL PROGRAM BOOK'

ISBN 0-9465-7614-9
90000
9 780946 576142

PHOENIX
PUBLISHING ASSOCIATES
£5.95